Learning Join Queries from User Examples

Learning Join Queries from User Examples

ANGELA BONIFATI, University of Lyon 1, University of Lille 1, Inria LINKS
RADU CIUCANU, University of Oxford, University of Lille 1, Inria LINKS
SŁAWEK STAWORKO, University of Lille 3, Inria LINKS, University of Edinburgh, DIACHRON, LFCS

We investigate the problem of learning join queries from user examples. The user is presented with a set of candidate tuples and is asked to label them as *positive* or *negative* examples, depending on whether or not she would like the tuples as part of the join result. The goal is to quickly infer an arbitrary n-ary join predicate across an arbitrary number m of relations while keeping the number of user interactions as minimal as possible. We assume no prior knowledge of the integrity constraints across the involved relations. Inferring the join predicate across multiple relations when the referential constraints are unknown may occur in several applications such as data integration, reverse engineering of database queries, and schema inference. In such scenarios, the number of tuples involved in the join is typically large. We introduce a set of strategies that let us inspect the search space and aggressively prune what we call "uninformative" tuples, and directly present to the user the *informative* ones i.e., those that allow to quickly find the goal query that the user has in mind. In this paper, we focus on the inference of joins with equality predicates and we also allow disjunctive join predicates and projection in the queries. We precisely characterize the frontier between tractability and intractability for the following problems of interest in these settings: consistency checking, learnability, and deciding the informativeness of a tuple. Next, we propose several strategies for presenting tuples to the user in a given order that lets minimize the number of interactions. We show the efficiency of our approach through an experimental study on both benchmark and synthetic datasets.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Query formulation.*

General Terms: Algorithms, Theory.

Additional Key Words and Phrases: SQL query discovery, reverse engineering, incomplete schema.

ACM Reference Format:
Angela Bonifati, Radu Ciucanu, and Sławek Staworko. 2015. Learning Join Queries from User Examples. *ACM Trans. Datab. Syst.* V, N, Article A (January YYYY), 38 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The amount of data and the number of available data sources continue to grow at an ever astounding rate allowing the users to satisfy more and more complex information needs. However, expressing complex information needs requires the use of formalisms for querying and integrating data sources that are typically mastered by only a small group of adept users. In real life, casual users often have to combine raw data coming from disparate data sources, with little or no knowledge of metadata and/or querying formalisms. Such unqualified users need to resort to brute force solutions of manipulating the data by hand. While there may exist providers of integrated data, the users may be unsatisfied with the quality of their results.

The problem of *assisting non-expert users to specify their queries* has been recently raised by [Jagadish et al. 2007; Nandi and Jagadish 2011]. More concretely, they have

Authors' addresses: angela.bonifati@univ-lyon1.fr, radu.ciucanu@cs.ox.ac.uk, sstawork@inf.ed.ac.uk.

DOI: http://dx.doi.org/10.1145/0000000.0000000

	From	*To*	*Airline*	*City*	*Discount*	
	Paris	Lille	AF	NYC	AA	(1)
	Paris	Lille	AF	Paris	None	(2)
+	Paris	Lille	AF	Lille	AF	(3)
+	Lille	NYC	AA	NYC	AA	(4)
	Lille	NYC	AA	Paris	None	(5)
	Lille	NYC	AA	Lille	AF	(6)
	NYC	Paris	AA	NYC	AA	(7)
−	NYC	Paris	AA	Paris	None	(8)
	NYC	Paris	AA	Lille	AF	(9)
	Paris	NYC	AF	NYC	AA	(10)
	Paris	NYC	AF	Paris	None	(11)
	Paris	NYC	AF	Lille	AF	(12)

Fig. 1. Integrated table.

observed that "constructing a database query is often challenging for the user, commonly takes longer than the execution of the query itself, and does not use any insights from the database".

Nevertheless, join specification may become feasible for non-expert users whenever they can easily access data and metadata altogether. This happens in traditional query specification paradigms, such as query-by-example [Zloof 1975], that are typically centered around a single database. When it comes to consider raw data coming from different data sources, such paradigms are not applicable any longer. The reason is twofold: (i) such data may not carry pertinent metadata to be able to specify a join predicate and (ii) value-based matching of tuples is unfeasible in most cases, due to a massive number of tuples.

In this paper, we consider very simple user input via Boolean membership queries ("Yes/No") to assist unfamiliar users to write their queries upon integrated data. In particular, we focus on two fundamental operators of any data integration or querying tool: *equijoins* – combining data from multiple sources, and *semijoins* – filtering data from one source based on the data from another source. Besides data integration, such operators are sensible in many other applications, such as reverse engineering of database queries and constraint inference in case of limited knowledge of the database schemas. In particular, the queries that we investigate are of practical use in the context of denormalized databases having a small number of relations with large numbers of attributes.

Inference algorithms for schema mappings have been recently studied in [Alexe et al. 2011a; 2011b] by leveraging data examples. However, such examples are expected to be provided by an expert user, namely the mapping designer, who is also responsible of selecting the mappings that best fit them. Query learning for relational queries with quantifiers has recently been addressed in [Abouzied et al. 2013; Abouzied et al. 2012]. There, the system starts from an initial formulation of the query and refines it based on primary-foreign key relationships and the input from the user. We discuss in detail the differences with our work at the end of this section. To the best of our knowledge, ours is the first work that considers inference of joins via simple tuple labeling and with no knowledge of integrity constraints.

Consider a scenario where a user working for a travel agency wants to build a list of flight&hotel packages. The user is not acquainted with querying languages and can access the information on flights and hotels in a denormalized table, result of some data integration scenario, as in Figure 1.

The airline operating every flight is known and some hotels offer a discount when paired with a flight of a selected airline. Two queries can be envisioned: one that selects travel packages consisting of a flight and a stay in a hotel and another one that additionally ensures that the package is combined in a way allowing a discount. These two queries correspond to the following equijoin predicates:

$$To = City, \quad (Q_1)$$
$$To = City \wedge Airline = Discount. \quad (Q_2)$$

Note that since we assume no knowledge of the schema and of the integrity constraints, a number of other queries can possibly be formulated but we remove them from consideration for the sake of simplicity and clarity of the example.

While the user may be unable to formulate her query, it is reasonable to assume that she can indicate whether or not a given pair of flight and hotel is of interest to her. We view this as labeling with $+$ and $-$ the tuples from the integrated table (Figure 1). For instance, suppose the user chooses the flight from Paris to Lille operated by Air France (AF) and the hotel in Lille. This corresponds to labeling by $+$ the tuple (3).

Observe that both queries Q_1 and Q_2 are consistent with this labeling i.e., both queries select the tuple (3). Naturally, the objective is to use the labeling of further tuples to identify the goal query i.e., the query that the user has in mind. Not every tuple can however serve this purpose. For instance, if the user labels next the tuple (4) with $+$, both queries remain consistent. Intuitively, the labeling of the tuple (4) does not contribute any new information about the goal query and is therefore *uninformative*, an important concept that we formalize in this paper. Since the input table may contain a large number of tuples, it may be unfeasible for the user to label every tuple.

For such a reason, we aim at limiting the number of tuples that the user needs to label in order to infer the goal query. More precisely, in this paper we propose solutions that analyze and measure the potential information about the goal query that labeling a tuple can contribute and present to the user tuples that maximize this measure. In particular, since uninformative tuples do not contribute any additional information, they would not be presented to the user. In the example of the flight&hotel packages, a tuple whose labeling can distinguish between Q_1 and Q_2 is, for instance, the tuple (8) because Q_1 selects it and Q_2 does not. If the user labels the tuple (8) with $-$, then the query Q_2 is returned; otherwise Q_1 is returned. We also point out that the use of only *positive* examples, tuples labeled with $+$, is not sufficient to identify all possible queries. As an example, query Q_2 is contained in Q_1, and therefore, satisfies all positive examples that Q_1 does. Consequently, the use of *negative* examples, tuples with label $-$, is necessary to distinguish between these two.

The foundations of such an interactive scenario for the inference of join queries have been studied in [Bonifati et al. 2014a]. However, the problem setting considered there is restricted to two relations on which only conjunctions of equality predicates can be learned. This paper substantially extends the results of [Bonifati et al. 2014a], by allowing (i) an arbitrary number of relations in the join inference and (ii) by adding disjunction to the join predicates.

First, notice that the extension to an arbitrary number of relations is rather natural. For instance, in our motivating example one can consider multiple flights and multiple hotels e.g., a user may be interested in a round trip with a stay in a hotel in an intermediate city. We have considered such queries in a synthetic scenario in the experimental evaluation.

Second, to illustrate a case when the disjunction is useful, assume for instance that the user is interested in travel packages consisting of a flight and a stay in a hotel (either in the source or destination city), combined in a way allowing a discount. This

query corresponds to the following disjunction of conjunctions of equijoin predicates:

$$(\mathit{From} = \mathit{City} \wedge \mathit{Airline} = \mathit{Discount}) \vee (\mathit{To} = \mathit{City} \wedge \mathit{Airline} = \mathit{Discount}).$$

To infer such a query, the user has to label on the instance from Figure 1 the tuples (3) and (7) as positive examples, and the tuples (8) and (11) as negative examples.

Summarizing, the *main contributions* of our paper are the following:

— We characterize the *learnability* of join queries using a definition based on the standard framework of *language identification in the limit with polynomial time and data* [Gold 1967; 1978]. Essentially, we show that the equijoins are learnable (with or without disjunction), while the semijoins are learnable only when disjunction is allowed. In particular, to prove that the semijoins without disjunction are not learnable, we have used the intractability of the consistency checking, a fundamental problem underlying learning i.e., to decide whether there exists a query consistent with a given set of examples.
Thus, we precisely characterize the frontier between the learnable and the non learnable cases depending on whether or not we allow disjunction and/or projection. We point out that all these learnability results are novel w.r.t. previous work [Bonifati et al. 2014a]. Moreover, consistency checking has been only studied in that work for the simple case of two relations and without disjunction, and, additionally, the intractability proof for consistency checking in the case of semijoins was missing.
— We focus on an interactive scenario inspired by the well-known framework of *learning with membership queries* [Angluin 1988], we characterize the potential information that labeling a given tuple may contribute to the join inference process, and identify uninformative tuples. More precisely, we propose two notions of uninformativeness, one based on the knowledge of the goal query and one not based on this knowledge, and we show that the two notions are equivalent.
Then, we prove that for all aforementioned learnable cases, deciding whether a tuple is informative can be tested in polynomial time. This is a non-trivial generalization of a result given in [Bonifati et al. 2014a] only for two relations and without disjunction. Additionally, we show that this problem remains intractable for semijoins without disjunction, which is again a novel contribution w.r.t. [Bonifati et al. 2014a].
— We propose a set of strategies for interactively inferring a goal join query and we show their efficiency within an experimental study on both TPC-H and synthetic data. The experimental study significantly improves the analysis done in previous work [Bonifati et al. 2014a] as here we consider all the queries of the TPC-H benchmark while they focused only on a small subset of the benchmark, limited to simple joins on two tables. The queries that we report in this paper take in fact into account an arbitrary number of tables. Then, to cope with the absence of disjunction in the TPC-H queries, we have defined a set of synthetic queries using such operator and implemented a synthetic dataset inspired by our motivating example. As a consequence, all the empirical results on learning disjunctive joins are novel w.r.t. previous work [Bonifati et al. 2014a].

Since our goal is to minimize the number of interactions with the user, our research is of interest for novel database applications e.g., query processing using the crowd [Franklin et al. 2011], where minimizing the number of interactions entails lower financial costs. In particular, crowdsourced joins have been mainly defined in terms of entity resolution, where joining two datasets means finding all pairs of tuples that refer to the same entity [Marcus et al. 2011; Wang et al. 2013]. Conversely, our goal is to handle arbitrary n-ary join predicates, thus targeting a quite different and more intricate goal for the crowd i.e., inferring such join predicates from a set of positive and negative labels.

Moreover, our research also applies to schema mapping inference, assuming a less expert user than in [Alexe et al. 2011a; 2011b]. Indeed, in our case the annotations correspond to simple membership queries [Angluin 1988] to be answered even by a user who is not familiar with schema mappings.

Organization

In Section 2, we introduce some preliminary notions. In Section 3, we define a framework for learning join queries from a given set of examples and analyze the complexity of two fundamental problems of interest: consistency checking and learnability. In Section 4, we describe the studied interactive scenario and investigate the problem of deciding the informativeness of a tuple. In Section 5, we propose practical strategies of presenting tuples to the user, while in Section 6, we experimentally evaluate their performance. Finally, we summarize the conclusions and outline directions of future work in Section 7.

Related work

A wealth of research on using computational learning theory [Kearns and Vazirani 1994] has been recently conducted in databases [Abouzied et al. 2013; Bex et al. 2010; Bonifati et al. 2014a; Bonifati et al. 2015; Lemay et al. 2010; Staworko and Wieczorek 2012; ten Cate et al. 2013]. Very recently, algorithms for learning relational queries (e.g., quantifiers [Abouzied et al. 2013], joins [Bonifati et al. 2014a]), XML queries (e.g., tree patterns [Staworko and Wieczorek 2012]), or graph queries (e.g., regular path queries [Bonifati et al. 2015]) have been proposed. Besides learning queries, researchers have investigated the learnability of relational schema mappings [ten Cate et al. 2013], as well as schemas [Bex et al. 2010] and transformations [Lemay et al. 2010] for XML. In this section, we discuss the positioning of our own work w.r.t. these and other papers.

Our work follows a very recent line of research on the inference of relational queries [Bonifati et al. 2014a; Zhang et al. 2013; Tran et al. 2009; Das Sarma et al. 2010]. As already mentioned above, we significantly generalize [Bonifati et al. 2014a] since we consider settings where we additionally allow the disjunction and an arbitrary number of relations. [Zhang et al. 2013] have focused on computing a join query starting from a database instance, its complete schema, and an output table. Clearly, their assumptions are different from ours. In particular, we do not assume any knowledge of the integrity constraints or the query result. In our approach, the latter has to be incrementally constructed via multiple interactions with the user, along with the join predicate itself. [Zhang et al. 2013] consider more expressive queries than we do, but when the integrity constraints are unknown, one can leverage our algorithms to yield those and apply their approach thereafter. Moreover, [Tran et al. 2009] have investigated the query by output problem: given a database instance, a query statement and its output, construct an instance-equivalent query to the initial statement. [Das Sarma et al. 2010] have studied the view definition problem i.e., given a database instance and a corresponding view instance, find the most succinct and accurate view definition. Both [Tran et al. 2009] and [Das Sarma et al. 2010] essentially use decision trees to classify tuples as selected or not selected in the query output or in the view output, respectively. We differ from their work in two ways: we do not know a priori the query output, and we need to discover it from user interactions; we have no initial query statement to start with.

The learnability definition that we employ in Section 3 is based on the standard framework of *language identification in the limit with polynomial time and data* [Gold 1967; 1978] adapted to learning join queries. Then, the interactive scenario studied

in Section 4 is inspired by the well-known framework of *learning with membership queries* [Angluin 1988].

A problem closely related to learning is *definability*. More precisely, [Bancilhon 1978] and [Paredaens 1978] have studied the decision problem, given a pair of relational instances, whether there exists a relational algebra expression which maps the first instance to the second one. Their research led to the notion of *BP-completeness*. Their results were later extended to the nested relational model [Van Gucht 1987] and to sequences of input-output pairs [Fletcher et al. 2009]. Learning and definability have in common the fact that they look for a query consistent with a set of examples. The difference is that learning allows the query to select or not the tuples that are not explicitly labeled as positive or negative examples while definability requires the query to select nothing else than the set of positive examples (i.e., all the other tuples are implicitly negative).

[Fan et al. 2011] have worked on discovering conditional functional dependencies using data mining techniques. We focus on simpler join constraints, and exploit an interactive scenario to discover them by interacting with the users.

Since our goal is to find the most informative tuples and ask the user to label them, our research is also related to the work of [Yan et al. 2013]. However, we do not consider keyword-based queries. Another work strongly related to ours has been done by [Abouzied et al. 2013; Abouzied et al. 2012], who have formalized a query learning model using membership questions [Angluin 1988]. They focus on learning quantified Boolean queries for the nested relational model and their main results are optimal algorithms for learning some subclasses of such queries [Abouzied et al. 2013] and a system that helps users specify quantifiers [Abouzied et al. 2012]. Primary-foreign key relationships between attributes are used to place quantified constraints and help the user tune her query, whereas we do not assume such knowledge. The goal of their system is somewhat different, in that their goal is to disambiguate a natural language specification of the query, whereas we focus on raw data to guess the "unknown" query that the user has in mind. The theoretical foundations of learning with membership queries have been studied in the context of schema mappings [ten Cate et al. 2013]. Moreover, [Alexe et al. 2011a; 2011b] have proposed a system which allows a user to interactively design and refine schema mappings via data examples. The problem of discovering schema mappings from data instances have been also studied in [Gottlob and Senellart 2010] and [Qian et al. 2012]. Our queries can be eventually seen as simple GAV mappings, even though our problem goes beyond data integration. Moreover, our focus is on proposing tuples to the user, while [Alexe et al. 2011a; 2011b] assume that an expert user chooses the data examples. Additionally, our notions of certain and uninformative tuples have connections with the approach of [Cohen and Weiss 2013] for XPath queries, even though joins are not considered there. Furthermore, our notion of entropy of a tuple is related to the work of [Sellam and Kersten 2013] on exploratory querying big data collections.

2. PRELIMINARIES

In this section, we define the basic concepts that we manipulate throughout the paper. Additionally, we summarize in Table I the notations used in the paper.

Relations. A *schema* $\mathscr{S}$ is a finite set of *relations* $\mathscr{S} = \{R_1, \ldots, R_m\}$ with implicitly given sets of *attributes* $attrs(R_i)$ (for $1 \leqslant i \leqslant m$). We assume that these sets of attributes are pairwise disjoint. We assume no other knowledge of the database schema, in particular no knowledge of the integrity constraints between the relations. Given a schema $\mathscr{S}$, a *signature* is a subset of relations $\mathcal{R} \subseteq \mathscr{S}$. We naturally extend $attrs$ to signatures i.e., given a signature $\mathcal{R}$, we have $attrs(\mathcal{R}) = \bigcup_{R \in \mathcal{R}} attrs(R)$.

Table I. Table of notations.

Symbol	*Meaning*
R	Relation
$\mathscr{S}$	Schema i.e., set of relations
$\mathcal{R} \subseteq \mathscr{S}$	Signature i.e., subset of the schema
$attrs(R)$ / $attrs(\mathcal{R})$	Set of attributes from a relation R / signature $\mathcal{R}$
$t : \{A_1, \ldots, A_k\} \to \mathcal{U}$	Tuple over attributes $A_1, \ldots, A_k$ and of domain $\mathcal{U}$
$sig(t) = \mathcal{R}$	Signature of tuple t, or alternatively, t is compatible with signature $\mathcal{R}$
$t_1 \cdot t_2$	Tuple of signature $\mathcal{R}_1 \cup \mathcal{R}_2$ (if t_1, t_2 are of disjoint signatures $\mathcal{R}_1, \mathcal{R}_2$)
$t[A]$	The value of the attribute A in t
$I(R)$	Instance of relation R i.e., a set of compatible tuples
$I(\mathcal{R})$	Instance of signature $\mathcal{R}$ i.e., union of all $I(R)$ for $R \in \mathcal{R}$
Ω	Set of all pairs of attributes from different relations in $\mathscr{S}$
$\theta \subseteq \Omega$	Join predicate
$\Theta \subseteq 2^{\Omega}$	Disjunctive join predicate
q	Join query
$q(I)$	Answers of query q over instance I
$\mathcal{R} = inSig(q)$	Input signature of q i.e., a nonempty subset of $\mathscr{S}$
$\mathcal{R}_o = outSig(q)$	Output signature of q i.e., a nonempty subset of $\mathcal{R}$
$\boldsymbol{Join}(\mathcal{R})$	Class of equijoins i.e, queries $(\mathcal{R}, \mathcal{R}, \theta)$
$\boldsymbol{Join}^{\ltimes}(\mathcal{R}, \mathcal{R}_o)$	Class of semijoins i.e, queries $(\mathcal{R}, \mathcal{R}_o, \theta)$ with $\mathcal{R}_o \subset \mathcal{R}$
$\boldsymbol{UJoin}(\mathcal{R})$	Class of disjunctive equijoins i.e, queries $(\mathcal{R}, \mathcal{R}, \Theta)$
$\boldsymbol{UJoin}^{\ltimes}(\mathcal{R}, \mathcal{R}_o)$	Class of disjunctive semijoins i.e, queries $(\mathcal{R}, \mathcal{R}_o, \Theta)$ with $\mathcal{R}_o \subset \mathcal{R}$
$D(\mathcal{R}, I)$	Cartesian product of instances of relations in $\mathcal{R}$
$(\bowtie_\theta \mathcal{R})(I)$	Answers of equijoin $(\mathcal{R}, \mathcal{R}, \theta)$ over instance I
$(\mathcal{R}_o \ltimes_\theta (\mathcal{R} \backslash \mathcal{R}_o))(I)$	Answers of semijoin $(\mathcal{R}, \mathcal{R}_o, \theta)$ over instance I
$(\bowtie_\Theta \mathcal{R})(I)$	Answers of disjunctive equijoin $(\mathcal{R}, \mathcal{R}, \Theta)$ over instance I
$(\mathcal{R}_o \ltimes_\Theta (\mathcal{R} \backslash \mathcal{R}_o))(I)$	Answers of disjunctive semijoin $(\mathcal{R}, \mathcal{R}_o, \Theta)$ over instance I
(t, α)	Example (positive if α is $+$ or negative if α is –)
S	Sample i.e., set of examples
$K = (\mathcal{R}, \mathcal{R}_o, Q)$	Learning setting (in signature $\mathcal{R}$, out signature $\mathcal{R}_o$, query class $\mathcal{Q}$)
Join	Class of settings $(\mathcal{R}, \mathcal{R}, \boldsymbol{Join}(\mathcal{R}))$ for learning equijoins
Join$^{\ltimes}$	Class of settings $(\mathcal{R}, \mathcal{R}_o, \boldsymbol{Join}^{\ltimes}(\mathcal{R}, \mathcal{R}_o))$ for learning semijoins
UJoin	Class of settings $(\mathcal{R}, \mathcal{R}, \boldsymbol{UJoin}(\mathcal{R}))$ for learning disj. equijoins
UJoin$^{\ltimes}$	Class of settings $(\mathcal{R}, \mathcal{R}_o, \boldsymbol{UJoin}^{\ltimes}(\mathcal{R}, \mathcal{R}_o))$ for learning disj. semijoins
$\mathcal{I}^K$	Set of all tuples compatible with relations in $\mathcal{R}$
$\mathcal{E}^K$	Set of all examples compatible with $\mathcal{R}_o$
$\mathcal{L}^K$	Function mapping every query and instance to its set of examples
$\mathrm{CONS}_{\mathsf{K}}$	Consistency checking for a class of settings K
$T(t)$ / $T(X)$	Most specific join predicate selecting tuple t / set of tuples X
$C^K(I, S)$	Set of all consistent queries w.r.t. instance I and sample S in setting K
$Uninf^K(I, S)$	Set of all uninformative examples w.r.t. I and S in K
$Cert^K(I, S)$	Set of all certain examples w.r.t. I and S in K

Instances. We assume an infinite *domain* $\mathcal{U}$ that is a set of numerical constants with equality $=$ and inequality $\neq$ defined in the natural way. Then, a *tuple* t over a set of attributes $\{A_1, \ldots, A_k\}$ is a function $t : \{A_1, \ldots, A_k\} \to \mathcal{U}$ that associates a value of the domain to each attribute. Moreover, given a tuple $t : \{A_1, \ldots, A_k\} \to \mathcal{U}$, if there exists a signature $\mathcal{R} \subseteq \mathscr{S}$ such that $\bigcup_{R \in \mathcal{R}} attrs(R) = \{A_1, \ldots, A_k\}$, we say that $\mathcal{R}$ is the *signature of* t i.e., $sig(t) = \mathcal{R}$, or alternatively, that t is *compatible* with $\mathcal{R}$.

Given two tuples t_1 and t_2 over disjoint signatures $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively, by $t_1 \cdot t_2$ we denote the tuple t over the signature $\mathcal{R}_1 \cup \mathcal{R}_2$ such that $t[A] = t_1[A]$ for every attribute A from $attrs(\mathcal{R}_1)$ and $t[A] = t_2[A]$ for every attribute A from $attrs(\mathcal{R}_2)$. Further-

more, an *instance* of a schema $\mathscr{S}$ is a function that associates to each relation $R \in \mathscr{S}$ a finite set of tuples $\{t_1, \ldots, t_p\}$ such that $sig(t_i) = \{R\}$ (for $1 \leqslant i \leqslant p$). For each relation $R \in \mathscr{S}$, we denote its corresponding set of tuples by $I(R)$. Moreover, given a signature $\mathcal{R} \subseteq \mathscr{S}$, by $I(\mathcal{R}) = \bigcup_{R \in \mathcal{R}} I(R)$ we denote the set of tuples from I corresponding to all relations from $\mathcal{R}$ and we refer to it as the instance of $\mathcal{R}$.

Example 2.1. In this example, for simplicity reasons, we use a schema of two relations $\mathscr{S}_0 = \{R_1, R_2\}$ with $attrs(R_1) = \{A_1, A_2\}$ and $attrs(R_2) = \{B_1, B_2, B_3\}$. Moreover, take the following instance I that contains 4 tuples for R_1 and 3 tuples for R_2.

$I(R_1) =$

	A_1	A_2
t_1	0	1
t_2	0	2
t_3	2	2
t_4	1	0

$I(R_2) =$

	B_1	B_2	B_3
t'_1	1	1	0
t'_2	0	1	2
t'_3	2	0	0

Queries. A *query* is basically a function that takes an instance and returns a set of tuples. More formally, a query q has an *input signature* $inSig(q)$ that is a non-empty set of input relations and an *output signature* $outSig(q)$ that is a non-empty set of output relations that we assume being a subset of the input signature i.e., $outSig(q) \subseteq inSig(q)$. We say that a query q is over a schema $\mathscr{S}$, or alternatively, is *compatible* with $\mathscr{S}$ if $inSig(q) \subseteq \mathscr{S}$. Then, given a query q over a schema $\mathscr{S}$ and an instance I of $\mathscr{S}$, the *answers* to q over I, denoted $q(I)$, is a finite set of tuples of signature $outSig(q)$.

In this paper, we focus on four classes of *join queries* that we define in the remainder of this section. To this purpose, let us first define, for a schema $\mathscr{S}$, the set Ω such that

$$\Omega = \bigcup_{R, R' \in \mathscr{S}, R \neq R'} attrs(R) \times attrs(R').$$

Then, a *join predicate* is a subset $\theta \subseteq \Omega$. Moreover, a *disjunctive join predicate* is a union of join predicates i.e., a subset $\Theta \subseteq 2^{\Omega}$.

Classes of join queries. Given a schema $\mathscr{S}$, a *join query* q is essentially a triple that consists of a non-empty input signature $inSig(q) \subseteq \mathscr{S}$, a non-empty output signature $outSig(q) \subseteq inSig(q)$, and a (disjunctive) join predicate i.e., it can be of the form $(\mathcal{R}, \mathcal{R}_o, \theta)$ or $(\mathcal{R}, \mathcal{R}_o, \Theta)$, where $\mathcal{R} \subseteq \mathscr{S}$, $\mathcal{R}_o \subseteq \mathcal{R}$, $\mathcal{R} \neq \emptyset$, $\mathcal{R}_o \neq \emptyset$, and $\theta \subseteq \Omega$ or $\Theta \subseteq 2^{\Omega}$. Given a class of join queries $\mathcal{Q}$, we refer to $\mathcal{R}$ and $\mathcal{R}_o$ as the input signature $inSig(\mathcal{Q})$ and the output signature $outSig(\mathcal{Q})$ of the class of queries $\mathcal{Q}$, respectively.

Since we consider two possibilities for the relationship between $\mathcal{R}$ and $\mathcal{R}_o$ (they can differ or not) and two possibilities for the join predicates (they can be disjunctive or not), we obtain four classes of join queries that we define below, together with their semantics. To be able to formalize their semantics, we introduce first the *Cartesian product* of an instance I of a signature $\mathcal{R} \subseteq \mathscr{S}$ that is $D(\mathcal{R}, I) = \times_{R \in \mathcal{R}} I(R)$. In the rest of the paper, we model the notion of integrated table (cf. Introduction) with Cartesian product. Moreover, in the remainder we use the terms Cartesian product and integrated table interchangeably.

The considered classes of queries are the following:

(1) *Join*$(\mathcal{R})$ (*equijoins*) – queries of the form $(\mathcal{R}, \mathcal{R}, \theta)$, where $\mathcal{R} \subseteq \mathscr{S}$, $\mathcal{R} \neq \emptyset$, and $\theta \subseteq \Omega$. We always present such queries as $(\bowtie_\theta \mathcal{R})$. Given an instance I of $\mathscr{S}$, we have:

$$(\bowtie_\theta \mathcal{R})(I) = \{t \in D(\mathcal{R}, I) \mid \forall (A, A') \in \theta.\ t[A] = t[A']\}.$$

(2) *Join*$^{\ltimes}(\mathcal{R}, \mathcal{R}_o)$ (*semijoins*) – queries of the form $(\mathcal{R}, \mathcal{R}_o, \theta)$, where $\mathcal{R} \subseteq \mathscr{S}$, $\mathcal{R}_o \subset \mathcal{R}$, $\mathcal{R}_o \neq \emptyset$, and $\theta \subseteq \Omega$. We always present such queries as $(\mathcal{R}_o \ltimes_\theta (\mathcal{R} \backslash \mathcal{R}_o))$. Given an

instance I of $\mathscr{S}$, we have:

$$(\mathcal{R}_o \ltimes_\theta (\mathcal{R} \backslash \mathcal{R}_o))(I) = \{t \in D(\mathcal{R}_o, I) \mid \exists t' \in D(\mathcal{R} \backslash \mathcal{R}_o, I).\ \forall (A, A') \in \theta.\ (t \cdot t')[A] = (t \cdot t')[A']\}.$$

(3) *UJoin*$(\mathcal{R})$ (*disjunctive equijoins*) – queries of the form $(\mathcal{R}, \mathcal{R}, \Theta)$, where $\mathcal{R} \subseteq \mathscr{S}$, $\mathcal{R} \neq \varnothing$, and $\Theta \subseteq 2^\Omega$. We always present such queries as $(\bowtie_\Theta \mathcal{R})$. Given an instance I of $\mathscr{S}$, we have:

$$(\bowtie_\Theta \mathcal{R})(I) = \{t \in D(\mathcal{R}, I) \mid \exists \theta \in \Theta.\ \forall (A, A') \in \theta.\ t[A] = t[A']\}.$$

(4) *UJoin*$^\ltimes(\mathcal{R}, \mathcal{R}_o)$ (*disjunctive semijoins*) – queries of the form $(\mathcal{R}, \mathcal{R}_o, \Theta)$, where $\mathcal{R} \subseteq \mathscr{S}$, $\mathcal{R}_o \subset \mathcal{R}$, $\mathcal{R}_o \neq \varnothing$, and $\Theta \subseteq 2^\Omega$. We always present such queries as $(\mathcal{R}_o \ltimes_\Theta (\mathcal{R} \backslash \mathcal{R}_o))$. Given an instance I of $\mathscr{S}$, we have:

$$(\mathcal{R}_o \ltimes_\Theta (\mathcal{R} \backslash \mathcal{R}_o))(I) = \{t \in D(\mathcal{R}_o, I) \mid \exists t' \in D(\mathcal{R} \backslash \mathcal{R}_o, I).\ \exists \theta \in \Theta.\ \forall (A, A') \in \theta.\ (t \cdot t')[A] = (t \cdot t')[A']\}.$$

Notice that the four aforementioned classes of join queries correspond to four classes of relational algebra expressions [Abiteboul et al. 1995]:

(1) equijoins: $(\bowtie_\theta \mathcal{R}) = \bowtie_{\bigwedge_{(A,A') \in \theta, A \in attrs(R), A' \in attrs(R')} R[A] = R'[A']} (\mathcal{R})$,

(2) semijoins: $(\mathcal{R}_o \ltimes_\theta (\mathcal{R} \backslash \mathcal{R}_o)) = \Pi_{attrs(\mathcal{R}_o)}(\bowtie_\theta \mathcal{R})$,

(3) disjunctive equijoins: $(\bowtie_\Theta \mathcal{R}) = \bigcup_{\theta \in \Theta} (\bowtie_\theta \mathcal{R})$,

(4) disjunctive semijoins: $(\mathcal{R}_o \ltimes_\Theta (\mathcal{R} \backslash \mathcal{R}_o)) = \Pi_{attrs(\mathcal{R}_o)}(\bowtie_\Theta \mathcal{R})$.

When $\mathcal{R}$ and $\mathcal{R}_o$ differ, we say that the join result is *projected* on the relations of $\mathcal{R}_o$. Since a signature is a set of relations, we can either project on all attributes of a given relation or on none of them, hence our definition does not directly allow projecting only on a subset of the attributes of a relation. However, we point out that we can support such cases by applying the following reduction: given a relation R such that we want to project only on a subset of its attributes, it suffices to (i) vertically partition R in two relations R_1 and R_2 containing the attributes that we want and we do not want to project, respectively, and (ii) add the corresponding join condition between R_1 and R_2 to the goal join query.

Moreover, when the two signatures are known from the context, we often identify a join query by its (disjunctive) join predicate.

Example 2.1 *(continued).* We illustrate the four classes of join queries. Take the signatures $\mathcal{R} = \{R_1, R_2\}$, $\mathcal{R}_1 = \{R_1\}$, $\mathcal{R}_2 = \{R_2\}$, and the following join predicates:

$$\theta_1 = \{(A_1, B_1), (A_2, B_3)\}, \qquad \theta_2 = \{(A_2, B_2)\}, \qquad \theta_3 = \{(A_2, B_1), (A_2, B_2), (A_2, B_3)\}.$$

The answers over I to the queries defined by the aforementioned signatures and join predicates are:

$$\begin{aligned}
&(\bowtie_{\theta_1} \mathcal{R})(I) = \{t_2 \cdot t'_2, t_4 \cdot t'_1\}, && (\mathcal{R}_1 \ltimes_{\theta_1} \mathcal{R}_2)(I) = \{t_2, t_4\},\\
&(\bowtie_{\theta_2} \mathcal{R})(I) = \{t_1 \cdot t'_1, t_1 \cdot t'_2, t_4 \cdot t'_3\}, && (\mathcal{R}_1 \ltimes_{\theta_2} \mathcal{R}_2)(I) = \{t_1, t_4\},\\
&(\bowtie_{\theta_3} \mathcal{R})(I) = \varnothing, && (\mathcal{R}_1 \ltimes_{\theta_3} \mathcal{R}_2)(I) = \varnothing. \qquad \square
\end{aligned}$$

Next, consider the disjunctive join predicates

$$\Theta_1 = \{\{(A_1, B_1)\}, \{(A_2, B_3)\}\}, \quad \Theta_2 = \{\{(A_2, B_2)\}\}, \quad \Theta_3 = \{\{(A_2, B_1), (A_2, B_2)\}, \{(A_2, B_3)\}\}$$

The answers over I to the queries defined by the aforementioned signatures and disjunctive join predicates are:

$$(\bowtie_{\Theta_1} \mathcal{R})(I) = \{t_1 \cdot t'_2, t_2 \cdot t'_2, t_3 \cdot t'_2, t_3 \cdot t'_3, t_4 \cdot t'_1, t_4 \cdot t'_3\}, \quad (\mathcal{R}_1 \ltimes_{\Theta_1} \mathcal{R}_2)(I) = I(R_1),$$
$$(\bowtie_{\Theta_2} \mathcal{R})(I) = \{t_1 \cdot t'_1, t_1 \cdot t'_2, t_4 \cdot t'_3\}, \quad (\mathcal{R}_1 \ltimes_{\Theta_2} \mathcal{R}_2)(I) = \{t_1, t_4\},$$
$$(\bowtie_{\Theta_3} \mathcal{R})(I) = \{t_1 \cdot t'_1, t_2 \cdot t'_2, t_3 \cdot t'_2, t_4 \cdot t'_1, t_4 \cdot t'_3\}, \quad (\mathcal{R}_1 \ltimes_{\Theta_3} \mathcal{R}_2)(I) = I(R_1). \qquad \square$$

In the next sections, we study the problem of learning join queries from examples given by the user.

3. LEARNING JOIN QUERIES FROM A SET OF EXAMPLES

In this section, we study the problem of learning join queries from a given set of examples. First, we define a *learning framework* (Section 3.1). Then, we investigate the *consistency checking* problem (Section 3.2) that is fundamental for establishing our *learnability results* (Section 3.3).

3.1. Learning framework

Assume a schema $\mathscr{S}$. An *example* is a pair (t, α), where t is a tuple and $\alpha \in \{+, -\}$. We say that an example of the form $(t, +)$ is a *positive example* while an example of the form $(t, -)$ is a *negative example*. Moreover, recall that a tuple $t : \{A_1, \ldots, A_k\} \to \mathcal{U}$ is compatible with a signature $\mathcal{R} \subseteq \mathscr{S}$ if $\bigcup_{R \in \mathcal{R}} attrs(R) = \{A_1, \ldots, A_k\}$. Then, an example (t, α) is compatible with a signature $\mathcal{R}$ if t is compatible with $\mathcal{R}$. Also recall that by $D(\mathcal{R}, I)$ we denote the Cartesian product of the instance I of a signature $\mathcal{R} \subseteq \mathscr{S}$.

A *learning setting* is a tuple $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$ where

— $\mathcal{R} \subseteq \mathscr{S}$ is the non-empty input signature of K i.e., $inSig(K) = \mathcal{R}$,
— $\mathcal{R}_o \subseteq \mathcal{R}$ such that $\mathcal{R}_o \neq \varnothing$ is the non-empty output signature of K i.e., $outSig(K) = \mathcal{R}_o$,
— $\mathcal{Q}$ is a class of queries such that $inSig(Q) = \mathcal{R}$ and $outSig(Q) = \mathcal{R}_o$.

For instance, take two non-empty signatures $\mathcal{R} \subseteq \mathscr{S}$ and $\mathcal{R}_o \subset \mathcal{R}$. Then, the following are examples of learning settings.

— $(\mathcal{R}, \mathcal{R}, \textit{Join}(\mathcal{R}))$ for learning equijoins over $\mathcal{R}$,
— $(\mathcal{R}, \mathcal{R}_o, \textit{Join}^{\ltimes}(\mathcal{R}, \mathcal{R}_o))$ for learning semijoins over $\mathcal{R}$ and $\mathcal{R}_o$,
— $(\mathcal{R}, \mathcal{R}, \textit{UJoin}(\mathcal{R}))$ for learning disjunctive equijoins over $\mathcal{R}$,
— $(\mathcal{R}, \mathcal{R}_o, \textit{UJoin}^{\ltimes}(\mathcal{R}, \mathcal{R}_o))$ for learning disjunctive semijoins over $\mathcal{R}$ and $\mathcal{R}_o$.

We illustrate learning settings in Example 3.1.

Example 3.1. Recall our Flight&Hotel running example from the Introduction, whose Cartesian product is depicted in Figure 1. Assume that the first three attributes (*From, To, Airline*) come from a relation *Flight*, and the other two attributes (*City, Discount*) come from a relation *Hotel*. Take the following learning settings:

— $(\{\textit{Flight, Hotel}\}, \{\textit{Flight, Hotel}\}, \textit{Join}(\{\textit{Flight, Hotel}\}))$. This is basically the learning setting that we illustrated in the Introduction. We can learn equijoins between the two tables e.g., $\textit{To} = \textit{City} \wedge \textit{Airline} = \textit{Discount}$, which selects travel packages consisting of a flight and a hotel stay in the destination, combined in a way allowing a discount.
— $(\{\textit{Flight, Hotel}\}, \{\textit{Flight}\}, \textit{Join}^{\ltimes}(\{\textit{Flight, Hotel}\}, \{\textit{Flight}\}))$. In this setting, the examples are tuples from the table *Flight* i.e., the table on which the result of the join query is projected. We can learn the join predicate from the previous case, but with a different semantics: it selects the flights for which there exists a hotel stay in

the destination, combined in a way allowing a discount (we project away the information concerning the hotel).

— $(\{\textit{Flight, Hotel}\}, \{\textit{Flight, Hotel}\}, \textit{UJoin}(\{\textit{Flight, Hotel}\}))$. In this setting, the examples are tuples from the Cartesian product of the two relations, and the disjunction is allowed in the goal query. An example of query that can be learned in this setting is $(\textit{From} = \textit{City} \wedge \textit{Airline} = \textit{Discount}) \vee (\textit{To} = \textit{City} \wedge \textit{Airline} = \textit{Discount})$, which basically selects the travel packages consisting of a flight and a stay in a hotel (either in the source or destination city), combined in a way allowing a discount.

— $(\{\textit{Flight, Hotel}\}, \{\textit{Flight}\}, \textit{UJoin}^{\ltimes}(\{\textit{Flight, Hotel}\}, \{\textit{Flight}\}))$. In this setting, the examples are tuples from the table *Flight* and disjunction is permitted in the goal query. We can learn the join predicate from the previous case, but we now project away the information concerning the hotel. □

Next, we define four classes of learning settings i.e., one for each class of join queries.

— The class of settings for learning equijoins:

$$\mathsf{Join} = \{(\mathcal{R}, \mathcal{R}, \textit{Join}(\mathcal{R})) \mid \mathcal{R} \subseteq \mathscr{S} \text{ such that } \mathcal{R} \neq \varnothing \text{ for every schema } \mathscr{S}\}.$$

— The class of settings for learning semijoins:

$$\mathsf{Join}^{\ltimes} = \{(\mathcal{R}, \mathcal{R}_o, \textit{Join}^{\ltimes}(\mathcal{R}, \mathcal{R}_o)) \mid \mathcal{R} \subseteq \mathscr{S} \text{ and } \mathcal{R}_o \subset \mathcal{R} \text{ such that } \mathcal{R}_o \neq \varnothing \text{ for every schema } \mathscr{S}\}.$$

— The class of settings for learning disjunctive equijoins:

$$\mathsf{UJoin} = \{(\mathcal{R}, \mathcal{R}, \textit{UJoin}(\mathcal{R})) \mid \mathcal{R} \subseteq \mathscr{S} \text{ such that } \mathcal{R} \neq \varnothing \text{ for every schema } \mathscr{S}\}.$$

— The class of settings for learning disjunctive semijoins:

$$\mathsf{UJoin}^{\ltimes} = \{(\mathcal{R}, \mathcal{R}_o, \textit{UJoin}^{\ltimes}(\mathcal{R}, \mathcal{R}_o)) \mid \mathcal{R} \subseteq \mathscr{S} \text{ and } \mathcal{R}_o \subset \mathcal{R} \text{ such that } \mathcal{R}_o \neq \varnothing \text{ for every schema } \mathscr{S}\}.$$

Additionally, given a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$, we define:

— The set $\mathcal{I}^K = \bigcup_{R \in \mathcal{R}} \mathcal{U}^{attrs(R)}$ of all tuples compatible with relations in $\mathcal{R}$.
— The set $\mathcal{E}^K = \mathcal{U}^{attrs(\mathcal{R}_o)} \times \{+, -\}$ of all examples compatible with $\mathcal{R}_o$.
— The function $\mathcal{L}^K$ that maps every query q from $\mathcal{Q}$ and instance $I \subseteq \mathcal{I}^K$ such that $inSig(q) = sig(D(\mathcal{R}, I))$ to the corresponding set of examples $S \subseteq \mathcal{E}^K$ such that $outSig(q) = sig(D(\mathcal{R}_o, I))$ i.e., $\mathcal{L}^K(q, I) = q(I) \times \{+\} \cup (D(\mathcal{R}_o, I) \backslash q(I)) \times \{-\}$.

Given a learning setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$ and an instance $I \subseteq \mathcal{I}^K$, a *sample w.r.t. I and K* is a subset $S \subseteq \mathcal{E}^K$ of examples (t, α) such that $sig(t) = outSig(\mathcal{Q})$ and $\alpha \in \{+, -\}$. When it does not lead to confusion, we simply write that S is a *sample* or a *sample over I*. For a sample S, we denote the set of positive examples $\{t \in D(\mathcal{R}_o, I) \mid (t, +) \in S\}$ by S_+ and the set of negative examples $\{t \in D(\mathcal{R}_o, I) \mid (t, -) \in S\}$ by S_-.

Moreover, given a learning setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$, an instance $I \subseteq \mathcal{I}^K$, a sample $S \subseteq \mathcal{E}^K$ over I, and a query $q \in \mathcal{Q}$, we say that q is *consistent with S* if it selects all positive examples and none of the negative ones i.e., $S_+ \subseteq q(I)$ and $S_- \cap q(I) = \varnothing$. Naturally, the goal of learning should be the construction of a consistent join query.

When the class of queries is clear from the context, we may write that a (disjunctive) join predicate is consistent with a sample rather than the query that it defines. For instance, we may simply write that the join predicate θ is consistent with a sample S instead of writing that the query $(\bowtie_\theta \mathcal{R})$ is consistent with S.

Next, we propose a definition of learnability based on the standard framework of *language identification in the limit with polynomial time and data* [Gold 1967; 1978]

Table II. Summary of complexity results for consistency checking.

	Equijoins	*Semijoins*
Without disjunction	PTIME (Theorem 3.4)	NP-complete (Theorem 3.5)
With disjunction	PTIME (Theorem 3.8)	PTIME (Theorem 3.8)

adapted to learning join queries. A *learning algorithm* is an algorithm that takes an instance and a sample, and returns a query in $\mathcal{Q}$ or a special value *null*.

Definition 3.2. A class of queries $\mathcal{Q}$ is *learnable in polynomial time and data* in its corresponding learning setting $K = (inSig(\mathcal{Q}), outSig(\mathcal{Q}), \mathcal{Q})$ if there exists a *polynomial learning algorithm* $learner$ satisfying the following two conditions:

(1) **Soundness.** For every instance $I \subseteq \mathcal{I}^K$ and every sample $S \subseteq \mathcal{E}^K$ over I, the algorithm $learner(I, S)$ returns a query $q \in \mathcal{Q}$ that is consistent with S or a special *null* value if no such query exists.
(2) **Completeness.** For every query $q \in \mathcal{Q}$ there exists an instance $I \subseteq \mathcal{I}^K$ and a sample $CS_q \subseteq \mathcal{E}^K$ over I such that for every sample S that extends CS_q consistently with q i.e., $CS_q \subseteq S \subseteq \mathcal{L}^K(q, I)$, the algorithm $learner(I, S)$ returns a query equivalent to q. Furthermore, the size of CS_q is polynomially bounded by the size of the query.

The sample CS_q is called the *characteristic sample* for q w.r.t. $learner$ and K. For a learning algorithm there may exist many such samples. The definition requires that a characteristic sample exists. The soundness condition is a natural requirement while the completeness condition guarantees that the learning algorithm constructs the goal query from a sufficiently rich (but still polynomial) set of examples.

3.2. Consistency checking

As we have already pointed out in Definition 3.2, we aim at a polynomial learning algorithm that returns a query that selects all positive examples and none of the negative ones. To this purpose, we first investigate the consistency checking problem i.e., deciding whether such a query exists. This also permits to check whether the user who has provided the examples is honest, has not made any error, and therefore, has labeled the tuples consistently with some goal join query that she has in mind.

More formally, the *consistency checking for a class of learning settings* K is the following decision problem: given a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$ from K, an instance $I \subseteq \mathcal{I}^K$, and a sample $S \subseteq \mathcal{E}^K$ over I, decide whether there exists a query $q \in \mathcal{Q}$ that is consistent with the sample. Consistency checking is parametrized by the learning setting and the corresponding decision problem CONS_K is:

$$\{(K, I, S) \mid K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q}) \in \mathsf{K}, I \subseteq \mathcal{I}^K, S \subseteq \mathcal{E}^K. \exists q \in \mathcal{Q}.\ S_+ \subseteq q(I) \wedge S_- \cap q(I) = \varnothing\}.$$

We summarize in Table II the complexity of consistency checking for the considered classes of join queries and we present the proofs of these results in the rest of the section.

3.2.1. Consistency checking for equijoins. First, we show that in the case of equijoins the consistency checking has a simple solution that employs an elementary tool that we introduce next. Given a tuple $t \in D(\mathcal{R}, I)$, we define the *most specific join predicate selecting* t as follows:

$$T(t) = \{(A, A') \mid t[A] = t[A'] \wedge A \in attrs(R) \wedge A' \in attrs(R') \wedge R \neq R'\}.$$

Additionally, we extend T to sets of tuples $T(X) = \bigcap_{t \in X} T(t)$. Our interest in T follows from the observation that for a given set of tuples X, if θ is a join predicate selecting X, then $\theta \subseteq T(X)$.

	A_1	A_2	B_1	B_2	B_3	T
$t_1 \cdot t'_1$	0	1	1	1	0	$\{(A_1, B_3), (A_2, B_1), (A_2, B_2)\}$
$t_1 \cdot t'_2$	0	1	0	1	2	$\{(A_1, B_1), (A_2, B_2)\}$
$t_1 \cdot t'_3$	0	1	2	0	0	$\{(A_1, B_2), (A_1, B_3)\}$
$t_2 \cdot t'_1$	0	2	1	1	0	$\{(A_1, B_3)\}$
+ $t_2 \cdot t'_2$	0	2	0	1	2	$\{(A_1, B_1), (A_2, B_3)\}$
$t_2 \cdot t'_3$	0	2	2	0	0	$\{(A_1, B_2), (A_1, B_3), (A_2, B_1)\}$
$t_3 \cdot t'_1$	2	2	1	1	0	$\varnothing$
− $t_3 \cdot t'_2$	2	2	0	1	2	$\{(A_1, B_3), (A_2, B_3)\}$
$t_3 \cdot t'_3$	2	2	2	0	0	$\{(A_1, B_1), (A_2, B_1)\}$
+ $t_4 \cdot t'_1$	1	0	1	1	0	$\{(A_1, B_1), (A_1, B_2), (A_2, B_3)\}$
$t_4 \cdot t'_2$	1	0	0	1	2	$\{(A_1, B_2), (A_2, B_1)\}$
$t_4 \cdot t'_3$	1	0	2	0	0	$\{(A_2, B_2), (A_2, B_3)\}$

Fig. 2. The Cartesian product $R_1 \times R_2$, the value of T for each of its tuples, and sample S_0.

Example 2.1 *(continued).* In Figure 2 we present the Cartesian product $R_1 \times R_2$, the value of T for each tuple from it, and the sample S_0 such that $S_{0,+} = \{t_2 \cdot t'_2, t_4 \cdot t'_1\}$ and $S_{0,-} = \{t_3 \cdot t'_2\}$. The sample is consistent and the most specific consistent join predicate is $\theta_0 = \{(A_1, B_1), (A_2, B_3)\}$. Another consistent join predicate (but not the most specific) is $\theta'_0 = \{(A_1, B_1)\}$. On the other hand, the sample S'_0 such that $S'_{0,+} = \{t_1 \cdot t'_2, t_1 \cdot t'_3\}$ and $S'_{0,-} = \{t_3 \cdot t'_1\}$ is not consistent. □

We next show that a sample S is consistent with a join predicate iff $T(S_+)$ selects no negative example.

LEMMA 3.3. *Given a setting $K = (\mathcal{R}, \mathcal{R}, \mathit{Join}(\mathcal{R}))$ in* Join*, an instance $I \subseteq \mathcal{I}^K$, and a sample $S \subseteq \mathcal{E}^K$ over I, it holds that $(K, I, S) \in \mathrm{CONS}_{\mathsf{Join}}$ iff $S_- \cap (\bowtie_{T(S_+)} \mathcal{R})(I) = \varnothing$.*

PROOF. For the *if* part, since $T(S_+)$ selects all positive examples (by definition) and selects no negative tuple (by hypothesis) we infer that $T(S_+)$ is a predicate consistent with the sample.

For the *only if* part, assume that there exists a predicate θ selecting all positive examples and none of the negative ones. Since $T(S_+)$ is the most specific join predicate selecting all positive examples, $\theta \subseteq T(S_+)$, and since θ selects no negative example, neither does $T(S_+)$. Hence, $T(S_+)$ is also a join predicate consistent with the set of examples. □

Next, we use the above Lemma to show the tractability of the consistency checking for equijoins.

THEOREM 3.4. $\mathrm{CONS}_{\mathsf{Join}}$ *is in PTIME.*

PROOF. Lemma 3.3 gives us a necessary and sufficient condition for solving this problem i.e., testing whether the join predicate $T(S_+)$ selects any negative example. First, we point out that for a tuple $t \in D(\mathcal{R}, I)$, a simple algorithm computes $T(t)$ and implicitly $T(S_+)$ in polynomial time. Then, we have to check whether these queries select any negative example, which can be also easily done in polynomial time. □

3.2.2. Consistency checking for semijoins. Next, we show that for semijoins the fundamental decision problem of consistency checking is unfortunately intractable, even when we consider two relations only, as we state below.

THEOREM 3.5. $\mathrm{CONS}_{\mathsf{Join}^\times}$ *is NP-complete. The result holds even when the schema consists of two relations only.*

PROOF. To prove the membership of the problem to NP, we point out that a Turing machine guesses a join predicate θ, which has polynomial size in the size of the input. Then, we can easily check in polynomial time whether θ selects all positive examples and none of the negative ones.

Next, we prove the NP-hardness of $\mathrm{CONS}_{\mathsf{Join}^\ltimes}$ by reduction from 3SAT, known as being NP-complete. Given a formula $\varphi = c_1 \wedge \ldots \wedge c_k$ in 3CNF over the set of variables $\{x_1, \ldots, x_n\}$, we construct:

— A schema $\mathscr{S}_\varphi = \{R_\varphi, P_\varphi\}$, where $attrs(R_\varphi) = \{id_R, A_1, \ldots, A_n\}$ and $attrs(P_\varphi) = \{id_P, B_1^t, B_1^f, \ldots, B_n^t, B_n^f\}$, the input signature $\mathcal{R}_\varphi = \mathscr{S}_\varphi$, the output signature $\mathcal{R}_{o\varphi} = \{R_\varphi\}$.
— The learning setting $K\varphi = (\mathcal{R}_\varphi, \mathcal{R}_{o\varphi}, \textit{Join}^\ltimes(\mathcal{R}_\varphi, \mathcal{R}_{o\varphi}))$.
— The instance $I_\varphi(R_\varphi)$ of the relation R_φ that contains:
  — For $1 \leqslant i \leqslant k$, a tuple $t_{R,i}$ with $t_{R,i}[id_R] = c_i^+$ and $t_{R,i}[A_j] = j$ (for $1 \leqslant j \leqslant n$),
  — A tuple $t'_{R,0}$ with $t'_{R,0}[id_R] = X$ and $t'_{R,0}[A_j] = j$ (for $1 \leqslant j \leqslant n$),
  — For $1 \leqslant i \leqslant n$, a tuple $t'_{R,i}$ with $t_{Ri}[id_R] = x_i^-$ and $t'_{R,i}[A_j] = j$ (for $1 \leqslant j \leqslant n$).
— The instance $I_\varphi(P_\varphi)$ of the relation P_φ that contains:
  — For $1 \leqslant i \leqslant k$, let $x_{k_1}, x_{k_2}, x_{k_3}$ the variables used by c_k, with $k_1, k_2, k_3 \in \{1, \ldots, n\}$. Then, we have a tuple for each of them, let it $t_{P,il}$, with $l \in \{1, 2, 3\}$ such that $t_{P,il}[id_P] = c_i^+$, and for $1 \leqslant j \leqslant n$:

$$\begin{cases} t_{P,il}[B_j^t] = t_{P,il}[B_j^f] = j \text{ if } j \neq k_l, \\ t_{P,il}[B_j^t] = j,\ t_{P,il}[B_j^f] = \perp \text{ if } j = k_l \text{ and } x_{k_l} \text{ is a positive literal}, \\ t_{P,il}[B_j^t] = \perp,\ t_{P,il}[B_j^f] = j \text{ otherwise}. \end{cases}$$

  — A tuple $t'_{P,0}$ such that $t'_{P,0}[id_P] = Y$ and $B_i^t = B_i^f = i$ (for $1 \leqslant i \leqslant n$).
  — For $1 \leqslant i \leqslant n$, a tuple $t'_{P,i}$ such that $t'_{P,i}[id_P] = x_i^-$ and $t'_{P,i}[B_j^t] = t'_{P,i}[B_j^f] = j$ if $i \neq j$ or $t'_{P,i}.[B_j^t] = t'_{P,i}[B_j^f] = \perp$ otherwise (for $1 \leqslant j \leqslant n$).
— The sample $S_\varphi \subseteq R_\varphi \times \{+, -\}$ such that $S_{\varphi,+} = \{t_{R,1}, \ldots, t_{R,k}\}$ and $S_{\varphi,-} = \{t'_{R,0}, t'_{R,1}, \ldots t'_{R,n}\}$.

For example, for $\varphi_0 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$, we construct $I_{\varphi_0}(R_{\varphi_0})$ and $I_{\varphi_0}(P_{\varphi_0})$, respectively as shown below:

$I_{\varphi_0}(R_{\varphi_0})$ =

	id_R	A_1	A_2	A_3	A_4
$t_{R,1}$	c_1^+	1	2	3	4
$t_{R,2}$	c_2^+	1	2	3	4
$t'_{R,0}$	X	1	2	3	4
$t'_{R,1}$	x_1^-	1	2	3	4
$t'_{R,2}$	x_2^-	1	2	3	4
$t'_{R,3}$	x_3^-	1	2	3	4
$t'_{R,4}$	x_4^-	1	2	3	4

$I_{\varphi_0}(P_{\varphi_0})$ =

	id_P	B_1^t	B_1^f	B_2^t	B_2^f	B_3^t	B_3^f	B_4^t	B_4^f
$t_{P,11}$	c_1^+	1	$\perp$	2	2	3	3	4	4
$t_{P,12}$	c_1^+	1	1	$\perp$	2	3	3	4	4
$t_{P,13}$	c_1^+	1	1	2	2	3	$\perp$	4	4
$t_{P,21}$	c_2^+	$\perp$	1	2	2	3	3	4	4
$t_{P,22}$	c_2^+	1	1	2	2	$\perp$	3	4	4
$t_{P,23}$	c_2^+	1	1	2	2	3	3	4	$\perp$
$t'_{P,0}$	Y	1	1	2	2	3	3	4	4
$t'_{P,1}$	x_1^-	$\perp$	$\perp$	2	2	3	3	4	4
$t'_{P,2}$	x_2^-	1	1	$\perp$	$\perp$	3	3	4	4
$t'_{P,3}$	x_3^-	1	1	2	2	$\perp$	$\perp$	4	4
$t'_{P,4}$	x_4^-	1	1	2	2	3	3	$\perp$	$\perp$

and the sample S_{φ_0} such that $S_{\varphi_0,+} = \{t_{R,1}, t_{R,2}\}$ and $S_{\varphi_0,-} = \{t'_{R,0}, t'_{R,1}, t'_{R,2}, t'_{R,3}, t'_{R,4}\}$.

We claim that φ is satisfiable iff $(K_\varphi, I_\varphi, S_\varphi) \in \mathrm{CONS}_{\mathsf{Join}^\ltimes}$.

For the *if* part, let θ_0 the join predicate that selects all positive examples and none of the negatives. We observe from the instances of R_φ and P_φ that $\theta_0 \subseteq \{(id_R, id_P)\} \cup \{(A_i, B_i^t), (A_i, B_i^f) \mid 1 \leqslant i \leqslant n\}$. Because $S_{\varphi,-}$ is not empty, we infer that θ_0 is not empty. Moreover, we show that θ_0 contains (id_R, id_P) and at least one of (A_i, B_i^t) and (A_i, B_i^f) (for $1 \leqslant i \leqslant n$) by eliminating the other cases:

(1) If we assume that $(id_R, id_P) \notin \theta_0$, we infer that the negative example $t'_{R,0}$ belongs to $(\mathcal{R}_{o\varphi} \ltimes_{\theta_0} (\mathcal{R}_{o\varphi} \backslash \mathcal{R}_\varphi)(I)$, which contradicts the fact that θ_0 is consistent with S_φ.
(2) If we assume that there exists an $1 \leqslant i \leqslant n$ such that neither (A_i, B_i^t) nor (A_i, B_i^f) belongs to θ_0, we infer that the negative example $t'_{R,i}$ belongs to $(\mathcal{R}_{o\varphi} \ltimes_{\theta_0} (\mathcal{R}_{o\varphi} \backslash \mathcal{R}_\varphi)(I)$, which contradicts the fact that θ_0 is consistent with S_φ.

Thus, we know that (id_R, id_P) belongs to θ_0 and at least one of (A_i, B_i^t) and (A_i, B_i^f) also belongs to θ_0 (for $1 \leqslant i \leqslant n$). From the construction of the instance we infer that there exists a join between A_i and B_i^v (with $v \in \{t, f\}$) if the valuation encoded in v for x_i does not make false the clause whose number is encoded in id_P (for $1 \leqslant i \leqslant n$). Moreover, θ_0 is consistent with S_φ implies that for each tuple $t_{R,i}$ from R (with $1 \leqslant i \leqslant k$), there exists a tuple $t_{P,il}$ in P (with $1 \leqslant l \leqslant 3$) such that $t_{R,i}[id_R] = t_{P,il}[id_P]$ and for $1 \leqslant j \leqslant n$ there exists $v \in \{t, f\}$ such that $t_{R,i}[A_j] = t_{P,il}[B_j^v]$. Thus, the valuation encoded in the B_j^v's from θ_0 (for $1 \leqslant j \leqslant n$) satisfies φ.

For the *only if* part, take the valuation $V : \{x_1, \ldots, x_n\} \to \{true, false\}$ that makes φ true. We construct the join predicate θ_0 that contains (id_R, id_P) and $(A_i, B_i^{v_i})$ where $v_i \in \{t, f\}$ corresponds to the valuation $V(x_i)$. From the construction of $I_\varphi(P_\varphi)$, we infer that for $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant k$, there exists $1 \leqslant l \leqslant 3$ such that $t_{P,jl}[id_P] = c_j^+$ and $t_{P,jl}[B_i^{v_i}] = i$. We infer that θ_0 is consistent with S_φ, and therefore, $(K_\varphi, I_\varphi, S_\varphi) \in \mathrm{CONS}_{\mathsf{Join}^\ltimes}$.

Clearly, the described reduction works in polynomial time. □

3.2.3. Consistency checking for disjunctive equijoins and semijoins. We show that consistency checking is tractable when adding the disjunction, both for equijoins and semijoins. We start with the disjunctive semijoins and then we point out that the disjunctive equijoins enjoy the same computational properties since they are in fact a particular case. Let us first present a necessary and sufficient condition for a sample to be consistent.

Table III. Summary of learnability results.

	Equijoins	*Semijoins*
Without disjunction	Yes (Theorem 3.9)	No (Theorem 3.10)
With disjunction	Yes (Theorem 3.11)	Yes (Theorem 3.11)

LEMMA 3.6. *Given a setting* $K = (\mathcal{R}, \mathcal{R}_o, \textit{UJoin}^{\ltimes}(\mathcal{R}, \mathcal{R}_o))$ *in* $\mathsf{UJoin}^{\ltimes}$*, an instance* $I \subseteq \mathcal{I}^K$*, and a sample* $S \subseteq \mathcal{E}^K$ *over* I*, it holds that*

$$(K, I, S) \in \mathrm{CONS}_{\mathsf{UJoin}^{\ltimes}} \textit{ iff } \forall t \in S_+.\ \exists t' \in D(\mathcal{R}\backslash\mathcal{R}_o, I).\ \forall t'' \in S_-.\ t'' \notin (\mathcal{R}_o \ltimes_{T(t\cdot t')} (\mathcal{R}\backslash\mathcal{R}_o))(\mathcal{I}).$$

PROOF. For the *if* part, we construct a disjunctive join predicate Θ as follows. We start with $\Theta = \varnothing$ and for each $t \in S_+$ we take a $t' \in D(\mathcal{R}\backslash\mathcal{R}_o, I)$ such that $S_- \cap (\mathcal{R}_o \ltimes_{T(t\cdot t')} (\mathcal{R}\backslash\mathcal{R}_o))(\mathcal{I}) = \varnothing$ (we know by hypothesis that such a t' does exist) and we add $T(t \cdot t')$ to Θ. Notice that the constructed Θ selects all positive examples and none of the negative ones, hence we infer that Θ is a disjunctive predicate consistent with the sample.

For the *only if* part, assume that there exists a disjunctive join predicate Θ selecting all positive examples and none of the negative ones. Since Θ selects all positive examples, we infer that for every $t \in S_+$ there exists a predicate $\theta \in \Theta$ such that $t \in (\mathcal{R}_o \ltimes_\theta (\mathcal{R}\backslash\mathcal{R}_o))(\mathcal{I})$ and $S_- \cap (\mathcal{R}_o \ltimes_\theta (\mathcal{R}\backslash\mathcal{R}_o))(\mathcal{I}) = \varnothing$. This implies that for every $t \in S_+$ there exists a predicate $\theta \in \Theta$ and a tuple $t' \in D(\mathcal{R}\backslash\mathcal{R}_o, I)$ such that $t \cdot t' \in (\bowtie_\theta \mathcal{R})(I)$ and $S_- \cap (\mathcal{R}_o \ltimes_\theta (\mathcal{R}\backslash\mathcal{R}_o))(\mathcal{I}) = \varnothing$. Since for all such tuples t and t' the above θ is included in the most specific predicate $T(t \cdot t')$ and θ selects no negative, we infer that neither does $T(t \cdot t')$, which concludes the proof. □

In the particular case where the input and output signatures coincide, all t' from Lemma 3.6 are in fact empty tuples and Lemma 3.6 reduces to the following result.

COROLLARY 3.7. *Given a setting* $K = (\mathcal{R}, \mathcal{R}, \textit{UJoin}(\mathcal{R}))$ *in* UJoin*, an instance* $I \subseteq \mathcal{I}^K$*, and a sample* $S \subseteq \mathcal{E}^K$ *over* I*, it holds that*

$$(K, I, S) \in \mathrm{CONS}_{\mathsf{UJoin}} \textit{ iff } S_- \cap (\bowtie_{\bigcup_{t\in S_+}\{T(t)\}} \mathcal{R})(\mathcal{I}) = \varnothing.$$

Next, we show that consistency checking can be solved in polynomial time for disjunctive equijoins and semijoins.

THEOREM 3.8. $\mathrm{CONS}_{\mathsf{UJoin}}$ *and* $\mathrm{CONS}_{\mathsf{UJoin}^{\ltimes}}$ *are in PTIME.*

PROOF. Lemma 3.6 gives us a necessary and sufficient condition for solving the consistency checking for disjunctive semijoins. First, we point out that for two tuples $t \in S_+$ and $t' \in D(\mathcal{R}\backslash\mathcal{R}_o, I)$, a simple algorithm computes $T(t\cdot t')$ in polynomial time and this has to be done for $|S_+| \times |D(\mathcal{R}\backslash\mathcal{R}_o, I)|$ tuples. Then, we have to check whether these queries select any negative example, which can be also easily done in polynomial time. The aforementioned simple procedure can be also applied for disjunctive equijoins i.e., in the particular case where all t' are empty tuples. □

3.3. Learnability results

In this section, we use the developments from the previous section to characterize the learnability of different classes of join queries. We summarize in Table III the learnability results and we present the proofs in the rest of the section.

First, we show that the equijoins are learnable.

THEOREM 3.9. *The equijoins are learnable in polynomial time and data i.e., in settings from* Join.

PROOF. Take a setting $K = (\mathcal{R}, \mathcal{R}, \textit{Join}(\mathcal{R}))$ in Join. Then, take an instance $I \subseteq \mathcal{I}^K$ and a sample $S \subseteq \mathcal{E}^K$ over I. A simple polynomial algorithm outputs $T(S_+)$ if the

sample is consistent or *null* otherwise. Recall that this can be decided in polynomial time due to Theorem 3.4.

To show the completeness, we point out the construction, for every setting $K = (\mathcal{R}, \mathcal{R}, \textit{Join}(\mathcal{R}))$ in Join, for every query in $\textit{Join}(\mathcal{R})$, of an instance and of a polynomial characteristic sample $CS \subseteq \mathcal{E}^K$. Take a query $(\bowtie_\theta\ \mathcal{R})$ in $\textit{Join}(\mathcal{R})$. Then, we need an instance $I \subseteq \mathcal{I}^K$ having a tuple t such that $T(t) = \theta$. The construction of such an instance is straightforward. Assume that $\mathcal{R} = \{R_1, \ldots, R_n\}$. Then, we need that each $I(R_i)$ contains a tuple t_i (for $1 \leqslant i \leqslant n$) and let $t = t_1 \cdot \ldots \cdot t_n$. To ensure that $T(t) = \theta$, we fill the values of attributes of the t_i's as follows: (i) for each equivalence class in θ assign the same value to all respective attributes (but a different fresh value for each equivalence class), and (ii) for each attribute not involved in equalities in θ assign a different fresh value. Then, $CS = \{(t, +)\}$. In practice, a characteristic sample can appear on arbitrarily larger instances than the one that we used to illustrate its existence. Actually, as long as it exists a tuple t such that $T(t) = \theta$, and the current sample S contains $(t, +)$ and extends CS consistently with the goal query, then our learning algorithm is guaranteed to return the goal query. □

Next, we show that the intractability of the consistency checking for semijoins implies that the semijoins are not learnable.

THEOREM 3.10. *The semijoins are not learnable in polynomial time and data i.e., in settings from* $\mathsf{Join}^{\ltimes}$. *The result holds even when the schema consists of two relations only.*

PROOF. According to Definition 3.2, a learning algorithm should return *null* in polynomial time if a query consistent with the sample does not exist. Since checking the consistency of a sample is NP-complete (cf. Theorem 3.5), such an algorithm does not exist, hence the semijoins are not learnable in polynomial time and data. □

Finally, we show that both disjunctive equijoins and disjunctive semijoins are learnable.

THEOREM 3.11. *The disjunctive equijoins and disjunctive semijoins are learnable in polynomial time and data i.e., in settings from* UJoin *and* $\mathsf{UJoin}^{\ltimes}$*, respectively.*

PROOF. We show the soundness and completeness for the class of settings $\mathsf{UJoin}^{\ltimes}$ and we point out that the same algorithm is applicable for the class of settings UJoin that is a particular case where the input and output signatures coincide. Take a setting $K = (\mathcal{R}, \mathcal{R}_o, \textit{UJoin}^{\ltimes}(\mathcal{R}, \mathcal{R}_o))$ in $\mathsf{UJoin}^{\ltimes}$, an instance $I \subseteq \mathcal{I}^K$, and a sample $S \subseteq \mathcal{E}^K$ over I. If the sample is not consistent (decidable in polynomial time due to Theorem 3.8), the learning algorithm returns *null*. If the sample is consistent, the learning algorithm returns a consistent disjunctive join predicate constructed as follows: (i) start with an empty Θ, (ii) for every $t \in S_+$, for every $t' \in D(\mathcal{R} \backslash \mathcal{R}_o, I)$, if $T(t \cdot t')$ selects no negative example, add $T(t \cdot t')$ to Θ (we know that such tuples t' exist for every t due to Lemma 3.6), (iii) eliminate the redundant predicates from Θ and return $\Theta' = \{\theta \in \Theta \mid \nexists \theta' \in \Theta.\ \theta' \subset \theta\}$.

For example, if after (ii) we have $\Theta_0 = \{\{(A, B)\}, \{(A, B), (C, D)\}, \{(E, F)\}\}$, the predicate $\{(A, B), (C, D)\}$ is redundant because every tuple selected by it is also selected by $\{(A, B)\}$. Thus, by removing the redundant predicate we obtain $\Theta'_0 = \{\{(A, B)\}, \{(E, F)\}\}$.

To show the completeness, we point out the construction, for every setting $K = (\mathcal{R}, \mathcal{R}_o, \textit{UJoin}^{\ltimes}(\mathcal{R}, \mathcal{R}_o))$ in $\mathsf{UJoin}^{\ltimes}$, for every query in $\textit{UJoin}^{\ltimes}(\mathcal{R}, \mathcal{R}_o)$, of an instance and of a polynomial characteristic sample. Take a disjunctive join predicate Θ that defines a query $(\mathcal{R}_o \ltimes_\Theta (\mathcal{R} \backslash \mathcal{R}_o))$. First, we eliminate the redundant join predicates in

Θ and construct an equivalent disjunctive join predicate $\Theta' = \{\theta \in \Theta \mid \nexists \theta' \in \Theta.\ \theta' \subset \theta\}$. Then, we construct an instance $I \subseteq \mathcal{I}^K$ and a characteristic sample $CS \subseteq \mathcal{E}^K$ over I as follows:

— For every $\theta \in \Theta'$, there exists a tuple $t \in D(\mathcal{R}_o, I)$ and another $t' \in D(\mathcal{R} \backslash \mathcal{R}_o, I)$ such that $T(t \cdot t') = \theta$ and $t \in CS_+$.
— For every $\theta \in \Theta$, for every $\theta' \in \{\theta \backslash \{(A, A')\} \mid (A, A') \in \theta\}$, there exists a tuple $t \in D(\mathcal{R}_o, I)$ and another $t' \in D(\mathcal{R} \backslash \mathcal{R}_o, I)$ such that $T(t \cdot t') = \theta'$ and $t \in CS_-$.

To fill the both types of tuples, we use the same simple technique described in the proof of Theorem 3.9 i.e., (i) for each equivalence class in θ (or θ' for the second type) assign the same value to all respective attributes (but a different fresh value for each equivalence class), and (ii) for each attribute not involved in equalities in θ (or θ') assign a different fresh value.

The positive examples ensure that the simple algorithm described at the beginning of the proof retrieves from the instance each predicate from Θ' while the negative examples ensure that these predicates are indeed selected by the algorithm. To this purpose, each negative example encodes a more general join predicate that the actual one that we want the algorithm to select.

We end the proof by pointing out that the size of CS_+ is bounded by $|\Theta|$ while the size of CS_- is bounded by $|\Theta| \times \max_{\theta \in \Theta} |\theta|$, which means that the size of the characteristic sample is polynomial in the size of the goal Θ. □

4. LEARNING JOIN QUERIES FROM INTERACTIONS WITH THE USER

In this section, we study the problem of learning join queries from a different point of view, where we start with an empty sample that we enrich during the interactions with the user. First, we define our *interactive scenario* (Section 4.1). Then, we say a few words about the *instance-equivalent join queries* that may be output by the learning algorithm (Section 4.2). Moreover, we characterize the tuples that are *uninformative* or *informative* w.r.t. the learning process (Section 4.3).

4.1. Interactive scenario

Let us now consider the following *interactive scenario* of join query inference. Take a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$, an instance $I \subseteq \mathcal{I}^K$, and assume that the user has in mind a query that belongs to the class of queries $\mathcal{Q}$. The user is presented with a tuple from the Cartesian product $D(\mathcal{R}_o, I)$ and indicates whether the tuple is selected or not by the join query that she has in mind by labeling the tuple as a positive or negative example. This process is repeated until a sufficient knowledge of the goal join query has been accumulated (i.e., there exists at most one join query consistent with the user's labels).

This scenario is inspired by the well-known framework of *learning with membership queries* proposed by [Angluin 1988]. Especially with large instances, we would not like that the user has to label all the tuples of the integrated table, but only a small subset of them. Our goal is to minimize the number of interactions with the user while still being computationally efficient. In this context, an interesting question is choosing the right strategy of presenting tuples to the user. To answer this question, our approach leads through the analysis of the potential information that labeling a given tuple may contribute from the point of view of the inference process.

To this purpose, we first need to introduce some auxiliary notions. We assume the existence of some goal join query q^γ from $\mathcal{Q}$ and that the user labels the tuples in a manner consistent with q^γ. Furthermore, given an instance $I \subseteq \mathcal{I}^K$, we identify the sample $S^\gamma \subseteq \mathcal{E}^K$ over I corresponding to fully labeling the database instance:

$$S^\gamma_+ = q^\gamma(I) \quad \text{and} \quad S^\gamma_- = D(\mathcal{R}_o, I) \setminus q^\gamma(I).$$

Moreover, given a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$, an instance $I \subseteq \mathcal{I}^K$, and a sample $S \subseteq \mathcal{E}^K$ over I, we identify the set of all queries that are consistent with S over I in K:

$$\mathcal{C}^K(I, S) = \{q \in \mathcal{Q} \mid S_+ \subseteq q(I) \text{ and } S_- \cap q(I) = \varnothing.\}.$$

When K is clear from the context, we write simply $\mathcal{C}(I, S)$ instead of $\mathcal{C}^K(I, S)$. Initially, $S = \varnothing$, and hence, $\mathcal{C}(I, S) = \mathcal{Q}$. Because S is consistent with q^γ, the set $\mathcal{C}(I, S)$ always contains q^γ. Ideally, we would like to devise a strategy of presenting elements of $D(\mathcal{R}_o, I)$ to the user to get us "quickly" from $\varnothing$ to some S such that $\mathcal{C}(I, S) = \{q^\gamma\}$. Moreover, notice that for a consistent sample S and an unlabeled tuple t, the two possible labels of t split $\mathcal{C}(I, S)$ in two disjoint subsets. Formally, we have the following.

LEMMA 4.1. *Given a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$, an instance $I \subseteq \mathcal{I}^K$, a consistent sample $S \subseteq \mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o, I)$, it holds that*

$$\mathcal{C}(I, S) = \mathcal{C}(I, S \cup \{(t, +)\}) \cup \mathcal{C}(I, S \cup \{(t, -)\}) \quad \textit{and} \quad \mathcal{C}(I, S \cup \{(t, +)\}) \cap \mathcal{C}(I, S \cup \{(t, -)\}) = \varnothing.$$

PROOF. The subset $\mathcal{C}(I, S \cup \{(t, +)\}) \subseteq \mathcal{C}(I, S)$ is the set of queries in $\mathcal{C}(I, S)$ that select t while the subset $\mathcal{C}(I, S \cup \{(t, -)\}) \subseteq \mathcal{C}(I, S)$ is the set of queries in $\mathcal{C}(I, S)$ that do not select t. Notice that the intersection of the two subsets is empty and their union is $\mathcal{C}(I, S)$. □

4.2. Instance-equivalent join predicates

It is important to note that in practice the content of the instance I may not be rich enough to allow the exact identification of the goal query q^γ i.e., when $\mathcal{C}(I, S^\gamma)$ contains elements other than q^γ. In such a case, we want to return to the user a join query that is *equivalent to q^γ w.r.t. the instance I*, and hence, indistinguishable by the user.

For example, assuming that the goal query is a equijoin, we return to the user $T(S_+)$, which is equivalent to q^γ over the instance I i.e., $q^\gamma(I) = (\bowtie_{T(S_+)} \mathcal{R})(I)$, and hence indistinguishable by the user. To clarify when such a situation occurs, take the relations P_1, P_2 with the instance I below:

$$P_1 = \begin{array}{c|cc} & A_1 & A_2 \\ \hline t_1 & 1 & 1 \end{array} \qquad P_2 = \begin{array}{c|c} & B_1 \\ \hline t'_1 & 1 \end{array}$$

and the equijoin goal query q_1^γ defined by the join predicate $\theta_1 = \{(A_1, B_1)\}$. If we present the only tuple of the Cartesian product to the user, she labels it as a positive example, which yields the sample $S_1 = \{(t_1 \cdot t'_1, +)\}$. Then, $\mathcal{C}(I, S_1) = \textit{Join}(\{P_1, P_2\})$ and all its elements are equivalent to q_1^γ w.r.t. I. In particular, in this case we return to the user the join predicate $T(S_{1,+}) = \{(A_1, B_1), (A_2, B_1)\}$, where $\theta_1 \subsetneq T(S_{1,+})$.

Another situation when we return an instance-equivalent join query is when $q^\gamma(I)$ is empty, and therefore, the user labels all given tuples as negative examples. In such a case, we return to the user the most specific join query of that class. For example, if the goal query is an equijoin, we return the query defined by $T(S_+)$, which in this case equals Ω, which again is equivalent to q^γ over I.

4.3. Uninformative and informative tuples

In this section, we identify the tuples that do not yield new information when presented to the user. Before formally defining such tuples, we would like to intuitively illustrate the notion of (un)informativeness via Example 4.2.

Example 4.2. Recall our Flight&Hotel running example from the Introduction, whose Cartesian product is depicted in Figure 1. Assume that the first three attributes (*From, To, Airline*) come from a relation *Flight*, and the other two attributes

(*City, Discount*) come from a relation *Hotel*. Then, assume that we are in the setting $(\{$*Flight, Hotel*$\}, \{$*Flight, Hotel*$\}, \mathit{Join}(\{$*Flight, Hotel*$\}))$, where we want to learn an equijoin across the two relations.

If the user has labeled only the tuple (3), then notice that tuple (4) does not contribute any new information about the goal query. Indeed, tuples (3) and (4) are selected by precisely the same join predicates: $\varnothing$, *To=City*, *Airline=Discount*, and *To=City* $\wedge$ *Airline=Discount*. Thus, we know that regardless the query that the user had in mind when she labeled (3) as a positive example, she will also label (4) as a positive one (of course provided that the user is consistent with herself).

On the other hand, if the user has labeled only the tuple (3), then the tuple (8) is still informative for the learning process: if the user labels (8) as a positive example, we know that *To=City* $\wedge$ *Airline=Discount* is no longer a candidate join predicate because it does not select (8); conversely, if the user labels (8) as a negative example, we know that $\varnothing$ and *To=City* are no longer candidate join predicates because they both select the negative example (8). Hence, we observe that both labelings of tuple (8) permit us to filter the current set of candidate queries, in other words (8) is an informative tuple for the learning process. □

To formally define (un)informative tuples, take a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$ and assume that the user has in mind the goal query q^γ that belongs to the class of queries $\mathcal{Q}$. Let us assume for a moment that the goal q^γ is known. We say that an *example* (t, α) from S^γ is *uninformative* for the query q^γ w.r.t. an instance $I \subseteq \mathcal{I}^K$ and a sample $S \subseteq \mathcal{E}^K$ over I if $\mathcal{C}(I, S) = \mathcal{C}(I, S \cup \{(t, \alpha)\})$. In this case, we say that t is an *uninformative tuple* w.r.t. I and S. Formally, we define the set $\mathit{Uninf}^K(I, S)$ of all uninformative examples w.r.t. I and S in a setting K:

$$\mathit{Uninf}^K(I, S) = \{(t, \alpha) \in S^\gamma \mid \mathcal{C}^K(I, S) = \mathcal{C}^K(I, S \cup \{(t, \alpha)\})\}.$$

When K is clear from the context, we write simply $\mathit{Uninf}(I, S)$ instead of $\mathit{Unif}^K(I, S)$.

To illustrate the notion of uninformativeness, take the instance of the relations R_1 and R_2 from Example 2.1, the goal query q_0^γ defined by the join predicate $\{(A_2, B_3)\}$, and a sample S_0 such that $S_{0,+} = \{t_2 \cdot t'_2\}$ and $S_{0,-} = \{t_1 \cdot t'_3\}$. Notice that the examples $(t_4 \cdot t'_1, +)$ and $(t_2 \cdot t'_1, -)$ are uninformative.

Ideally, a smart inference algorithm should avoid presenting uninformative tuples to the user, but it is impossible to identify those tuples using the definition above without the knowledge of q^γ. This motivates us to introduce the notion of *certain tuples* w.r.t. an instance I and a sample S, which is independent of the goal query q^γ. Then, we prove that the notions of uninformative and certain tuples are equivalent and we identify the cases when testing the informativeness of a tuple can be done in polynomial time. We also mention that the notion of certain tuples is inspired by possible world semantics and certain answers [Imielinski and Lipski Jr. 1984] and already employed for XML querying for non-expert users by [Cohen and Weiss 2013]. Formally, we define the set $\mathit{Cert}^K(I, S)$ of all certain examples w.r.t. I and S in a setting K:

$$\begin{aligned}
\mathit{Cert}^K_+(I, S) &= \{t \in D(\mathcal{R}_o, I) \mid \forall q \in \mathcal{C}^K(I, S).\ t \in q(I)\},\\
\mathit{Cert}^K_-(I, S) &= \{t \in D(\mathcal{R}_o, I) \mid \forall q \in \mathcal{C}^K(I, S).\ t \notin q(I)\},\\
\mathit{Cert}^K(I, S) &= \mathit{Cert}^K_+(I, S) \times \{+\} \cup \mathit{Cert}^K_-(I, S) \times \{-\}.
\end{aligned}$$

When K is clear from the context, we write simply $\mathit{Cert}(I, S)$ instead of $\mathit{Cert}^K(I, S)$.

We assume w.l.o.g. that all samples that we manipulate are consistent. In case of an inconsistent sample S, we have $\mathcal{C}(I, S) = \varnothing$, in which case the notion of certain tuples

Table IV. Summary of complexity results for deciding the informativeness of a tuple.

	Equijoins	*Semijoins*
Without disjunction	PTIME (Theorem 4.5)	NP-complete (Theorem 4.6)
With disjunction	PTIME (Theorem 4.5)	PTIME (Theorem 4.5)

is of no interest. While the inclusion of $Cert(I,S)$ in $Uninf(I,S)$ is rather expected, we next show that the notions of uninformative and certain tuples are in fact equivalent.

LEMMA 4.3. *Given a setting $K=(\mathcal{R},\mathcal{R}_o,\mathcal{Q})$, an instance $I\subseteq\mathcal{I}^K$, and a consistent sample $S\subseteq\mathcal{E}^K$ over I, it holds that $Uninf(I,S)=Cert(I,S)$.*

PROOF. First, we show the inclusion $Uninf(I,S)\subseteq Cert(I,S)$. *Case 1.* Take a tuple t such that $(t,+)\in Uninf(I,S)$. From the definition of $\mathcal{C}$ we know that for every query q from $\mathcal{C}(I,S\cup\{(t,+)\})$ it holds that $t\in q(I)$. Because $\mathcal{C}(I,S)=\mathcal{C}(I,S\cup\{(t,+)\})$, we infer that for every query q from $\mathcal{C}(I,S)$ it holds that $t\in q(I)$, and therefore, $t\in Cert_+(I,S)$. *Case 2.* Take a tuple t such that $(t,-)\in Uninf(I,S)$. From the definition of $\mathcal{C}$ we know that for every query q from $\mathcal{C}(I,S\cup\{(t,-)\})$ it holds that $t\notin q(I)$. Because $\mathcal{C}(I,S)=\mathcal{C}(I,S\cup\{(t,-)\})$, we infer that for every query q from $\mathcal{C}(I,S)$ it holds that $t\notin q(I)$, and therefore, $t\in Cert_-(I,S)$.

Next, we prove the inclusion $Cert(I,S)\subseteq Uninf(I,S)$. *Case 1.* Take a tuple t in $Cert_+(I,S)$, which means that for every query q in $\mathcal{C}(I,S)$ it holds that $t\in q(I)$, which implies $\mathcal{C}(I,S)=\mathcal{C}(I,S\cup\{(t,+)\})$, hence $(t,+)\in Uninf(I,S)$. *Case 2.* Take a tuple t in $Cert_-(I,S)$, which means that for every query q in $\mathcal{C}(I,S)$ it holds that $t\notin q(I)$, which implies $\mathcal{C}(I,S)=\mathcal{C}(I,S\cup\{(t,-)\})$, in other words $(t,-)\in Uninf(I,S)$. □

Recall that we have defined the uninformative tuples w.r.t. the goal query and we have shown in Lemma 4.3 that $Uninf(I,S)=Cert(I,S)$, which means that we are able to characterize the uninformative tuples by using only the given sample, without having the knowledge of the goal query.

Next, let us also characterize the informative tuples i.e., those that contribute to the learning process. Take a setting $K=(\mathcal{R},\mathcal{R}_o,\mathcal{Q})$, an instance $I\subseteq\mathcal{I}^K$, a sample $S\subseteq\mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o,I)$. We say that t is *informative w.r.t. K, I, and S* if there does not exist a label $\alpha\in\{+,-\}$ such that $(t,\alpha)\in Uninf(I,S)$. When K, I, and S are clear from the context, we may write simply that t is *informative*. Next, we give a necessary and sufficient condition for a tuple to be informative.

LEMMA 4.4. *Given a setting $K=(\mathcal{R},\mathcal{R}_o,\mathcal{Q})$, an instance $I\subseteq\mathcal{I}^K$, a sample $S\subseteq\mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o,I)$, t is informative iff both $S\cup\{(t,+)\}$ and $S\cup\{(t,-)\}$ are consistent.*

PROOF. From Lemma 4.3 and the definition of uninformative tuples, we infer that given a label $\alpha\in\{+,-\}$, $(t,\alpha)\in Uninf(I,S)$ means $\mathcal{C}(I,S)=\mathcal{C}(I,S\cup\{(t,\alpha)\})$. By Lemma 4.1, this is equivalent to $\mathcal{C}(I,S\cup\{(t,\neg\alpha)\})=\varnothing$, which is furthermore equivalent to saying that the sample $S\cup\{(t,\neg\alpha)\}$ is not consistent. In other words t is uninformative iff there is a label $\alpha\in\{+,-\}$ such that $S\cup\{(t,\neg\alpha)\}$ is not consistent, which is equivalent to saying that t is informative iff both $S\cup\{(t,+)\}$ and $S\cup\{(t,-)\}$ are consistent. □

Consequently, we use the above characterization to analyze the complexity of deciding whether a tuple is informative or not. We present the summary of complexity results in Table IV and we prove them in the remainder.

First, we show that the tractability of the consistency checking implies the tractability of deciding the informativeness of a tuple. This result is in fact a generalization of a previous result presented in [Bonifati et al. 2014a], where only the setting of equijoins without disjunction has been studied.

THEOREM 4.5. *Given a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$ in* Join, UJoin, *or* UJoin$^{\ltimes}$, *an instance $I \subseteq \mathcal{I}^K$, a sample $S \subseteq \mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o, I)$, deciding whether t is informative is in PTIME.*

PROOF. If K is in Join, the result follows from Lemma 4.4 and Theorem 3.4. If K is in UJoin or UJoin$^{\ltimes}$, the result follows from Lemma 4.4 and Theorem 3.8. □

Next, we show that it is intractable to decide the informativeness of a tuple when the goal query is a semijoin.

THEOREM 4.6. *Given a setting $K = (\mathcal{R}, \mathcal{R}_o, \textit{Join}^{\ltimes}(\mathcal{R}, \mathcal{R}_o))$ in* Join$^{\ltimes}$, *an instance $I \subseteq \mathcal{I}^K$, a sample $S \subseteq \mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o, I)$, deciding whether t is informative is NP-complete. The result holds even when the schema consists of two relations only.*

PROOF. To prove the membership of the problem to NP, we point out that a Turing machine guesses a join predicate θ, which has polynomial size in the size of the input. Then, we can easily check in polynomial time whether θ is consistent with both $S \cup \{(t, +)\}$ and $S \cup \{(t, -)\}$.

To prove the NP-hardness, take the same reduction from the proof of Theorem 3.5. Then, add in the instance of P_φ, for every $1 \leqslant i \leqslant k$, one tuple t such that $t[id_P] = c_i^+$, $t[B_n^t] = t[B_n^f] = \perp'$, and for every $1 \leqslant j \leqslant n-1$, $t[B_j^t] = t[B_j^f] = j$. Moreover, we require $\perp \neq \perp'$. For the formula φ_0 from the proof of Theorem 3.5, add two tuples $(c_1^+, 1, 1, 2, 2, 3, 3, \perp', \perp')$ and $(c_2^+, 1, 1, 2, 2, 3, 3, \perp', \perp')$.

Then, consider S'_φ such that $S'_{\varphi,+} = \{t_{R,1}, \ldots, t_{R,k}\}$ and $S'_{\varphi,-} = \{t'_{R,0}, \ldots, t'_{R,n-1}\}$. We claim that φ is satisfiable iff the tuple $t'_{R,n}$ is informative. By Lemma 4.4, this is equivalent to saying that φ is satisfiable iff both $S'_\varphi \cup \{(t'_{R,n}, +)\}$ and $S'_\varphi \cup \{(t'_{R,n}, -)\}$ are consistent. Notice that $S'_\varphi \cup \{(t'_{R,n}, +)\}$ is clearly consistent since the join predicate $\{(id_R, id_P), (A_1, B_1^t), (A_1, B_1^f), \ldots, (A_{n-1}, B_{n-1}^t), (A_{n-1}, B_{n-1}^f)\}$ selects all positive and none of the negative tuples in $S'_\varphi \cup \{(t'_{R,n}, +)\}$. Thus, we have to prove that φ is satisfiable iff $S'_\varphi \cup \{(t'_{R,n}, -)\}$ is consistent, which follows exactly as in the proof of Theorem 3.5. □

We end this section by proposing additional characterizations of certain tuples that hold when the input and output signatures coincide and that are useful in practice (as we show later on in the paper when we discuss the lattice-based strategies). These additional characterizations are interesting because they are not stated in terms of consistency checking as in the general case (cf. Lemma 4.4). First, let us characterize the certain tuples for equijoins.

LEMMA 4.7. *Given a setting $K = (\mathcal{R}, \mathcal{R}, \textit{Join}(\mathcal{R}))$ in* Join, *an instance $I \subseteq \mathcal{I}^K$, a consistent sample $S \subseteq \mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o, I)$, it holds that:*

(1) *t belongs to $Cert_+(I, S)$ iff $T(S_+) \subseteq T(t)$,*
(2) *t belongs to $Cert_-(I, S)$ iff there exists a tuple t' in S_- such that $T(S_+) \cap T(t) \subseteq T(t')$.*

PROOF. *(1).* For the *if* part, assume $T(S_+) \subseteq T(t)$. From the definitions of $\mathcal{C}$ and T, we infer that for every θ defining a query $(\bowtie_\theta \mathcal{R})$ in $\mathcal{C}(I, S)$, it holds that $\theta \subseteq T(S_+)$, which furthermore, implies that $\theta \subseteq T(t)$, hence $t \in (\bowtie_\theta \mathcal{R})(I)$, in other words $t \in Cert_+(I, S)$.

For the *only if* part, assume $t \in Cert_+(I, S)$, which means that for every q in $\mathcal{C}(I, S)$ it holds that $t \in q(I)$. From the definitions of $\mathcal{C}$ and T, we infer that $(\bowtie_{T(S_+)} \mathcal{R})$ belongs to $\mathcal{C}(I, S)$, and therefore, $t \in (\bowtie_{T(S_+)} \mathcal{R})(I)$, which yields $T(S_+) \subseteq T(t)$.

(2). For the *if* part, take a tuple t' in S_- such that $T(S_+) \cap T(t) \subseteq T(t')$. This implies that for every q in $\mathcal{C}(I, S \cup \{(t,+)\})$ it holds that $t' \in q(I)$, hence $\mathcal{C}(I, S \cup \{(t,+)\}) = \varnothing$. By Lemma 4.1, we obtain $\mathcal{C}(I, S \cup \{(t,-)\}) = \mathcal{C}(I, S)$, which means that $(t,-) \in \mathit{Uninf}(I, S)$, and therefore, $t \in \mathit{Cert}_-(I, S)$ (by Lemma 4.3).

For the *only if* part, assume by absurd that for every t' in S_- it holds that $T(S_+) \cap T(t) \nsubseteq T(t')$, which implies that the set $\mathcal{C}(I, S \cup \{(t,+)\})$ is non-empty, hence $\mathcal{C}(S) \neq \mathcal{C}(I, S \cup \{(t,-)\})$ (by Lemma 4.1). This implies that $(t,-) \notin \mathit{Uninf}(I, S)$ that is equivalent by Lemma 4.3 to $(t,-) \notin \mathit{Cert}(I, S)$, which contradicts the hypothesis. We conclude that there exists a tuple t' in S_- such that $T(S_+) \cap T(t) \subseteq T(t')$. □

Next, let us also characterize the certain tuples for disjunctive equijoins.

LEMMA 4.8. *Given a setting $K = (\mathcal{R}, \mathcal{R}, \mathit{UJoin}(\mathcal{R}))$ in* UJoin*, an instance $I \subseteq \mathcal{I}^K$, a consistent sample $S \subseteq \mathcal{E}^K$ over I, and an unlabeled tuple t from $D(\mathcal{R}_o, I)$, it holds that:*

(1) t belongs to $\mathit{Cert}_+(I, S)$ iff there exists a tuple $t' \in S_+$ such that $T(t') \subseteq T(t)$.
(2) t belongs to $\mathit{Cert}_-(I, S)$ iff there exists a tuple $t' \in S_-$ such that $T(t) \subseteq T(t')$.

PROOF. *(1).* For the *if* part, let t' be a tuple in S_+ such that $T(t') \subseteq T(t)$. From the definitions of $\mathcal{C}$ and T, we infer that for every disjunctive join predicate Θ such that $(\bowtie_\Theta \mathcal{R}) \in \mathcal{C}(I, S)$ there is a join predicate $\theta \in \Theta$ such that $\theta \subseteq T(t')$, hence $\theta \subseteq T(t)$. This implies that for every $q \in \mathcal{C}(I, S)$ it holds that $t \in q(I)$, in other words $t \in \mathit{Cert}_+(I, S)$.

For the *only if* part, assume by absurd that for every $t' \in S_+$ we have $T(t') \nsubseteq T(t)$. Then, take the query $(\bowtie_{\bigcup_{t' \in S_+} \{T(t')\}} \mathcal{R})$ that belongs to $\mathcal{C}(I, S)$ and that does not select t. Consequently, $t \notin \mathit{Cert}_+(I, S)$, which contradicts the hypothesis. Thus, we conclude that there is a tuple $t' \in S_+$ such that $T(t') \subseteq T(t)$.

(2). For the *if* part, let t' be the tuple in S_- such that $T(t) \subseteq T(t')$. Then, from the definitions of $\mathcal{C}$ and T, we infer that for every disjunctive join predicate Θ such that $(\bowtie_\Theta \mathcal{R}) \in \mathcal{C}(I, S)$, for every $\theta \in \Theta$ it holds that $\theta \nsubseteq T(t')$, hence $\theta \nsubseteq T(t)$. This implies that $t \notin q(I)$ for every $q \in \mathcal{C}(I, S)$, in other words $t \in \mathit{Cert}_-(I, S)$.

For the *only if* part, assume by absurd that for every $t' \in S_-$ we have $T(t) \nsubseteq T(t')$. This implies that $(\bowtie_{\bigcup_{t' \in S_+ \cup \{t\}} \{T(t')\}} \mathcal{R})$ belongs to $\mathcal{C}(I, S)$ and selects t at the same time, which contradicts the hypothesis that $t \in \mathit{Cert}_-(I, S)$. Thus, we conclude that there exists a tuple $t' \in S_-$ such that $T(t) \subseteq T(t')$. □

5. STRATEGIES

In this section, we use the developments from the previous sections to propose efficient strategies for interactively presenting tuples to the user. First, in Section 5.1, we introduce the *general interactive inference algorithm* and we claim that there exists an *optimal strategy* that is however exponential. Consequently, we propose several *efficient strategies* that we essentially classify in two categories: *local* and *lookahead*. More precisely, in Section 5.2 we show that when the input and output signatures coincide, we can use the notion of *lattice of join predicates* to efficiently *pre-process* an instance and to define the *local strategies*. Then, in Section 5.3 we propose the *lookahead strategies* that work for all settings, being based on the notion of *entropy* of a tuple that we develop there.

5.1. General interactive inference algorithm

Take a setting $K = (\mathcal{R}, \mathcal{R}_o, \mathcal{Q})$ and an instance $I \subseteq \mathcal{I}^K$. A *strategy* Υ is a function that takes as input a Cartesian product $D(\mathcal{R}_o, I)$ and a sample $S \subseteq \mathcal{E}^K$ over I, and returns a tuple t in $D(\mathcal{R}_o, I)$. The *general interactive inference algorithm* (Algorithm 1) consists of selecting a tuple w.r.t. a *strategy* Υ and asking the user to label it as a positive or

negative example; this process continues until the *halt condition* Γ is satisfied. The algorithm continuously verifies the consistency of the sample, if at any moment the user labels a tuple such that the sample becomes inconsistent, the algorithm raises an exception.

We have chosen to investigate strategies that ask the user to label informative tuples only because we aim to minimize the number of interactions. Therefore, the sample that we incrementally construct is always consistent and our approach does not yield any error in lines 6-7. In our approach, we choose the strongest halt condition i.e., to stop the interactions when there is no informative tuple left:

$$\Gamma := \forall t \in D(\mathcal{R}_o, I).\ \exists \alpha \in \{+, -\}.\ (t, \alpha) \in S \cup \mathit{Uninf}(I, S).$$

At the end of the interactive process, we return the most specific join query consistent with the examples provided by the user (cf. the characterizations from Section 3 for different variations of the goal query class). For instance, if our goal is to infer a join query without projection and disjunction (i.e., an equijoin), we return $\theta = T(S_+)$.

However, the *halt condition* Γ may be weaker in practice, as the user might decide to stop the interactive process at an earlier time if, for instance, she finds some intermediate most specific consistent query to be satisfactory.

Algorithm 1 General interactive inference algorithm.

Input: the Cartesian product $D(\mathcal{R}_o, I)$
Output: a join query consistent with the user's labels
Parameters: strategy Υ, halt condition Γ
1: **let** $S = \varnothing$
2: **while** $\neg\Gamma$ **do**
3: **let** $t = \Upsilon(D(\mathcal{R}_o, I), S)$
4: **query** the user about the label α for t
5: $S := S \cup \{(t, \alpha)\}$
6: **if** S is not consistent **then**
7: **error**
8: **return** the most specific query selecting all positive examples

An optimal strategy exists and can be built by employing the standard construction of a minimax tree [Russell and Norvig 2010]. While the exact complexity of the optimal strategy remains an open question, a straightforward implementation of minimax requires exponential time (and is in PSPACE), which unfortunately renders it unusable in practice. As a consequence, we propose next a number of time-efficient strategies that attempt to minimize the number of interactions with the user and that we have implemented for our experimental study. For comparison we also introduce the *random strategy* (RND) that at each step chooses randomly an informative tuple.

5.2. Settings where the input and output signatures coincide

When the input and output signatures coincide i.e., we have settings of the form $(\mathcal{R}, \mathcal{R}, \mathcal{Q})$, the space of all potential join predicates expressible over a given instance is captured by a *lattice of join predicates* (Section 5.2.1). This lattice permits to efficiently *pre-process* the given instance (Section 5.2.2) and also to design some simple yet effective strategies that we call *local* (Section 5.2.3). For ease of exposition of our algorithms, in the remainder we denote the Cartesian product $D(\mathcal{R}, I)$ simply by D.

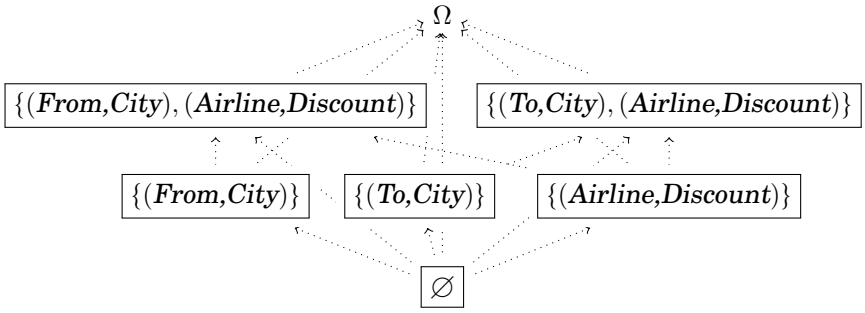


Fig. 3. Lattice of join predicates for the instance from Figure 1.

From	*To*	*Airline*	*City*	*Discount*	$T(t)$
Paris	Lille	AF	NYC	AA	$\varnothing$
Paris	Lille	AF	Paris	None	{(*From,City*)}
Paris	Lille	AF	Lille	AF	{(*To,City*), (*Airline,Discount*)}
NYC	Paris	AA	NYC	AA	{(*From,City*), (*Airline,Discount*)}
NYC	Paris	AA	Paris	None	{(*To,City*)}
Paris	NYC	AF	Lille	AF	{(*Airline,Discount*)}

Fig. 4. Result of pre-processing for the Cartesian product from Figure 1.

5.2.1. Lattice of join predicates. The *lattice of the join predicates* is $(\mathcal{P}(\Omega), \subseteq)$ with $\varnothing$ as its bottom-most node and Ω as its top-most node. We focus on *non-nullable* join predicates i.e., join predicates that select at least one tuple, because we expect the user to label at least one positive example during the interactive process. We also consider Ω in case the user decides to label all tuples as negative. Naturally, the number of non-nullable join predicates may still be exponential since all join predicates are non-nullable iff there exist two tuples $t \in R$ and $t' \in P$ such that $t[A_1] = \ldots = t[A_n] = t'[B_1] = \ldots = t'[B_m]$.

We present in Figure 3 the lattice of join predicates for the instance from Figure 1 and in Figure 5 the lattice corresponding to the instance from Example 2.1. For both of them, notice a set of non-nullable nodes and the Ω. We point out a correspondence between the nodes and the tuples in the Cartesian product D: a tuple $t \in D$ corresponds to a node of the lattice θ if $T(t) = \theta$. Not every node of the lattice has corresponding tuples and in Figure 5 only nodes in boxes have corresponding tuples (cf. Figure 2).

The main insight behind the lattice-based local strategies is the following: each label given by the user is propagated in the lattice using Lemma 4.7 if the goal query class consists of equijoins or using Lemma 4.8 if the goal query class consists of disjunctive equijoins. This allows us to *prune* parts of the lattice corresponding to the tuples that become uninformative. Basically, labeling a tuple t corresponding to a node θ as positive renders tuples corresponding to all nodes above θ uninformative and possibly some other nodes depending on tuples labeled previously (if the query class has no disjunction). Conversely, labeling t as negative prunes (at least) the part of the lattice below θ. For instance, take the lattice from Figure 5, assume an empty sample, and take the join predicate $\{(A_1, B_2), (A_1, B_3)\}$ and the corresponding tuple $t^\circ = t_1 \cdot t'_3$. If the user labels t° as a positive example, then the tuple $t_2 \cdot t'_3$ corresponding to $\{(A_1, B_2), (A_1, B_3), (A_2, B_1)\}$ becomes uninformative (cf. Lemma 4.7 or Lemma 4.8, respectively). On the other hand, if the user labels the tuple t° as a negative example, then the tuples $t_2 \cdot t'_1$ and $t_3 \cdot t'_1$ corresponding to $\{(A_1, B_3)\}$ and $\varnothing$ respectively, become uninformative (cf. the same two results). If we reason at the lattice level, the question "Which is the next tuple to

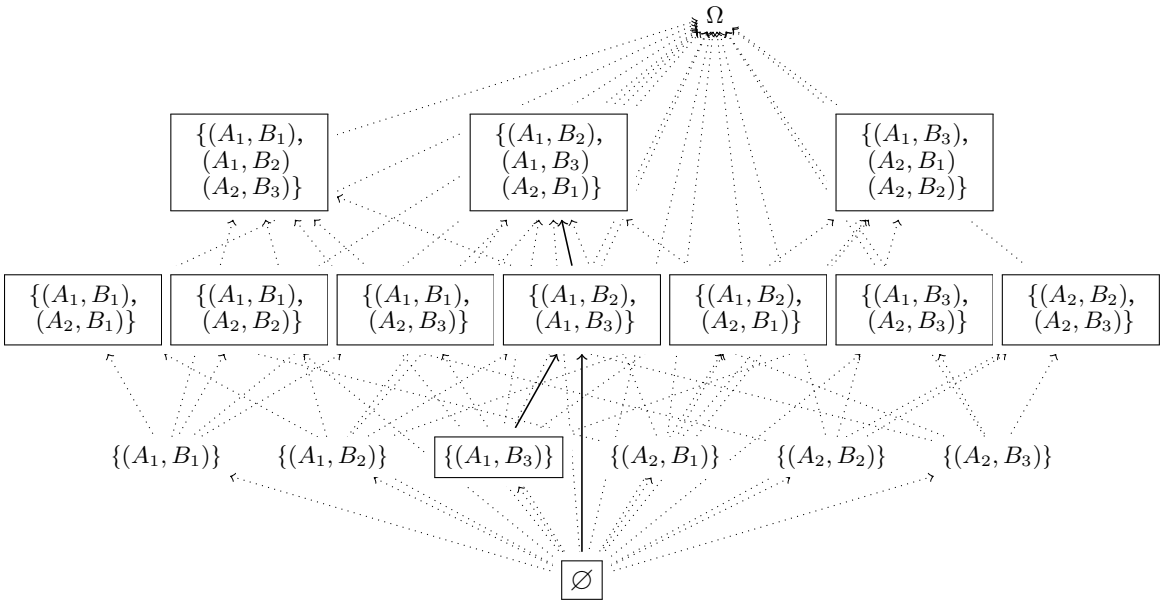


Fig. 5. Lattice of join predicates for the instance from Example 2.1.

present to the user?" intuitively becomes "Labeling which tuple allows us to prune as much of the lattice as possible?"

5.2.2. Pre-processing. We employ a simple mechanism to pre-process the input instance before asking the user to label any tuple. The idea is to remove redundant tuples based on the lattice of join predicates as follows: for each predicate θ such that there exists a tuple t in the Cartesian product D with $T(t) = \theta$, we take such a tuple t from the instance and we add it to the pre-processed instance.

For example, for the instance from Figure 1, we keep only one tuple for each predicate presented in a lattice node from Figure 3. Thus, we obtain the pre-processed instance from Figure 4, where we also present on the last column the corresponding $T(t)$ for each tuple. Notice that with this simple pre-processing procedure we have eliminated half of the tuples from the initial set, without altering the set of queries that can be learned on this instance. From now on, whenever we refer to an instance in the remainder, we mean in fact its pre-processed version where we have already eliminated all redundant tuples.

Even though we can easily imagine minor cases where pre-processing does not eliminate any tuple (e.g., for the instance from Example 2.1 whose lattice we depict in Figure 5), we have shown in the experimental study that it scales quite well for instances of billions of tuples that can be reduced to smaller sets of hundreds of tuples on which learning the goal query is clearly more efficient.

5.2.3. Local strategies. The principle behind the *local strategies* is that they propose tuples to the user following a simple order on the lattice. We call these strategies local because they do not take into account the quantity of information that labeling an informative tuple could bring to the inference process. As such, they differ from the lookahead strategies that we present in the next section. In this section we propose two local strategies, which essentially correspond to two basic variants of *navigating* in the lattice: the *bottom-up strategy* and the *top-down strategy*.

The *bottom-up strategy* (BU) (Algorithm 2) intuitively navigates the lattice of join predicates from the most general join predicate ($\varnothing$) towards the most specific one (Ω). It visits a minimal node of the lattice that has a corresponding informative tuple and asks the user to label it. If the label is positive, (at least) the part of the lattice above the node is pruned. If the label is negative, the current node is pruned (since the nodes below are not informative, they must have been pruned before). Recall the instance from Example 2.1 and its corresponding lattice in Figure 5. The BU strategy asks the user to label the tuple $t_0 = t_3 \cdot t'_1$ corresponding to $\varnothing$. If the label is positive, all nodes of the lattice are pruned and the empty join predicate returned. If the label is negative, the strategy selects the tuple $t_2 \cdot t'_1$ corresponding to the node $\theta_1 = \{(A_1, B_3)\}$ for labeling, etc. The BU strategy discovers quickly the goal join predicate $\varnothing$, but is inadequate to discover join predicates of bigger size. In the worst case, when the user provides only negative examples, the BU strategy might ask the user to label every tuple from the (pre-processed) Cartesian product.

Algorithm 2 Bottom-up strategy $\mathrm{BU}(D, S)$

1: **let** $m = \min(\{|T(t)| \mid t \in D \text{ such that } t \text{ is informative}\})$
2: **return** informative t such that $|T(t)| = m$

The *top-down strategy* (TD) (Algorithm 3) intuitively starts to navigate in the lattice of join predicates from the most specific join predicate (Ω) to the most general one ($\varnothing$). It has basically two behaviors depending on the contents of the current sample. First, when there is no positive example yet (lines 1-2), this strategy chooses a tuple t corresponding to a $\subseteq$-maximal join predicate i.e., whose $T(t)$ has no other non-nullable join predicate above it in the lattice (line 2). For example, for the instance corresponding to the lattice from Figure 5, we first ask the user to label the tuple corresponding to $\{(A_1, B_1), (A_1, B_2), (A_2, B_3)\}$, then the tuple corresponding to $\{(A_1, B_2), (A_1, B_3), (A_2, B_1)\}$, etc. Note that the relative order among these tuples corresponding to $\subseteq$-maximal join predicates is arbitrary. If the user labels all $\subseteq$-maximal join predicates as negative examples, we are able to infer the goal Ω without asking her to label all the Cartesian product (using Lemma 4.7 or Lemma 4.8, depending on whether or not disjunction is present in the goal query class). Thus, the TD strategy overcomes the mentioned drawback of the BU. On the other hand, if there is at least one positive example, then the goal join predicate is a non-nullable one, and the TD strategy turns into BU (lines 3-5). As we later show in Section 6, the TD strategy seems a good practical compromise.

Algorithm 3 Top-down strategy $\mathrm{TD}(D, S)$

1: **if** $S_+ = \varnothing$ **then**
2: **return** informative t such that $\nexists t' \in D.\ T(t) \subsetneq T(t')$
3: **else**
4: **let** $m = \min(\{|T(t)| \mid t \in D \text{ such that } t \text{ is informative}\})$
5: **return** informative t such that $|T(t)| = m$

	T	$u^+_{t,S}$	$u^-_{t,S}$	$entropy_S$
$t_1 \cdot t'_1$	$\{(A_1, B_3), (A_2, B_1), (A_2, B_2)\}$	0	2	(0,2)
$t_1 \cdot t'_2$	$\{(A_1, B_1), (A_2, B_2)\}$	0	1	(0,1)
$t_1 \cdot t'_3$	$\{(A_1, B_2), (A_1, B_3)\}$	1	2	**(1,2)**
$t_2 \cdot t'_1$	$\{(A_1, B_3)\}$	2	1	**(1,2)**
$t_2 \cdot t'_2$	$\{(A_1, B_1), (A_2, B_3)\}$	1	1	(1,1)
$t_2 \cdot t'_3$	$\{(A_1, B_2), (A_1, B_3), (A_2, B_1)\}$	0	4	(0,4)
$t_3 \cdot t'_1$	$\varnothing$	11	0	(0,11)
$t_3 \cdot t'_2$	$\{(A_1, B_3), (A_2, B_3)\}$	0	2	(0,2)
$t_3 \cdot t'_3$	$\{(A_1, B_1), (A_2, B_1)\}$	0	1	(0,1)
$t_4 \cdot t'_1$	$\{(A_1, B_1), (A_1, B_2), (A_2, B_3)\}$	0	2	(0,2)
$t_4 \cdot t'_2$	$\{(A_1, B_2), (A_2, B_1)\}$	1	1	(1,1)
$t_4 \cdot t'_3$	$\{(A_2, B_2), (A_2, B_3)\}$	0	1	(0,1)

Fig. 6. The Cartesian product corresponding to the instance from Example 2.1 and the entropy for each tuple, for an initial empty sample.

5.3. Lookahead strategies for general settings

In this section, we present the *lookahead strategies*. There are two key differences between them and the local strategies: (i) they are defined independently of the lattice of join predicates and can be thus employed in all considered settings (cf. Section 2); (ii) for all such settings, they take into account the *entropy* of an informative tuple i.e., the quantity of information that labeling that tuple could bring to the process of inference. More precisely, they continuously interleave the user's feedback and the inference process by taking into account the labels already given by the user to adjust the order of presenting new tuples for labeling. The notion of entropy that we develop here is in fact a generalization of a previous notion from [Bonifati et al. 2014a].

We need to introduce first some auxiliary notions. Given an informative tuple t from D and a sample S, let $u^\alpha_{t,S}$ be the number of tuples which become uninformative if the tuple t is labeled with α:

$$u^\alpha_{t,S} = |\mathit{Uninf}(I, S \cup \{(t, \alpha)\}) \setminus \mathit{Uninf}(I, S)|.$$

Now, the entropy of an informative tuple t w.r.t. a sample S, denoted $entropy_S(t)$, is the pair $(\min(u^+_{t,S}, u^-_{t,S}), \max(u^+_{t,S}, u^-_{t,S}))$, which captures the quantity of information that labeling the tuple t can provide. The entropy of uninformative tuples is undefined, however we never make use of it. In Figure 6 we present the entropy for each tuple from the Cartesian product of the instance from Example 2.1, for an empty sample.

Given two entropies $e = (a, b)$ and $e' = (a', b')$, we say that e *dominates* e' if $a \geqslant a'$ and $b \geqslant b'$. For example, $(1, 2)$ dominates $(1, 1)$ and $(0, 2)$, but it does not dominate $(2, 2)$ nor $(0, 3)$. Next, given a set of entropies E, we define the *skyline* of E, denoted $skyline(E)$, as the set of entropies e that are not dominated by any other entropy of E. For example, for the set of entropies of the tuples from Figure 6, the skyline is $\{(1, 2), (0, 11)\}$.

Next, we present the *one-step lookahead skyline strategy* ($\mathrm{L^1S}$) (Algorithm 4). We illustrate this strategy for the instance from Example 2.1, for an initial empty sample. First (line 1), we compute the entropy for each informative tuple from the Cartesian product. This corresponds to computing the last column from Figure 6. Then (line 2), we calculate the maximal value among all minimal values of the entropies computed at the previous step. For our example, this value is 1. Finally (lines 3-4), we return an informative tuple whose entropy is in the skyline of all entropies, and moreover, has as minimal value the number computed at the previous step. For our example, the skyline is $\{(1, 2), (0, 11)\}$, thus the entropy corresponding to the value computed previously (i.e., 1) is $(1, 2)$. Consequently, we return one of the tuples having the entropy $(1, 2)$, more

precisely either $t_1 \cdot t'_3$ or $t_2 \cdot t'_1$. Intuitively, according to $\mathrm{L}^1\mathrm{S}$ strategy, we choose to ask the user to label a tuple which permits to eliminate at least one and at most two additional tuples. Note that by $\min$ (resp. $\max$) we denote the minimal (resp. maximal) value from either a given set or a given pair of numbers, depending on the context.

Algorithm 4 One-step lookahead skyline $\mathrm{L}^1\mathrm{S}(D, S)$

1: **let** $E = \{entropy_S(t) \mid t \in D$ such that t is informative$\}$
2: **let** $m = \max(\{\min(e) \mid e \in E\})$
3: **let** e the entropy in $skyline(E)$ such that $\min(e) = m$
4: **return** informative t such that $entropy_S(t) = e$

The $\mathrm{L}^1\mathrm{S}$ strategy naturally extends to *k-steps lookahead skyline strategy* ($\mathrm{L}^k\mathrm{S}$). The difference is that instead of taking into account the quantity of information that labeling *one* tuple could bring to the inference process, we take into account the quantity of information for labeling k tuples. Note that if k is greater than the total number of informative tuples in the Cartesian product, then the strategy becomes optimal and thus inefficient. For such a reason, in the experiments we focus on a lookahead of two steps, which is a good trade-off between keeping a relatively low computation time and minimizing the number of interactions. Therefore, we present such strategy in the remainder.

Algorithm 5 $entropy^2_S(t)$

1: **for** $\alpha \in \{+, -\}$ **do**
2: **let** $S' = S \cup \{(t, \alpha)\}$
3: **if** $\nexists t' \in D$ such that t' is informative w.r.t. S' **then**
4: **let** $e_\alpha = (\infty, \infty)$
5: **continue**
6: **let** $E = \varnothing$
7: **for** $t' \in D$ s.t. t' is informative w.r.t. S' **do**
8: **let** $u^+ = |Uninf(I, S \cup \{(t, \alpha), (t', +)\}) \setminus Uninf(I, S)|$
9: **let** $u^- = |Uninf(I, S \cup \{(t, \alpha), (t', -)\}) \setminus Uninf(I, S)|$
10: $E := E \cup \{(\min(u^+, u^-), \max(u^+, u^-))\}$
11: **let** $m = \max(\{\min(e) \mid e \in E\})$
12: **let** e_α the entropy in $skyline(E)$ s.t. $\min(e_\alpha) = m$
13: **let** $m = \min(\{\min(e_+), \min(e_-)\})$
14: **return** e_α such that $\min(e_\alpha) = m$

We need to extend first the notion of entropy of a tuple to the notion of $entropy^2$ of a tuple. Given an informative tuple t and a sample S, the $entropy^2$ of t w.r.t. S, denoted $entropy^2_S(t)$, intuitively captures the minimal quantity of information that labeling t and another tuple can bring to the inference process. The construction of the $entropy^2$ is quite technical (Algorithm 5) and we present an example below. Take the sample $S = \{(t_1 \cdot t'_3, +), (t_3 \cdot t'_1, -)\}$. Note that $Uninf(I, S) = \{(t_2 \cdot t'_3, +)$, $(t_1 \cdot t'_2, -)$, $(t_2 \cdot t'_2, -)$, $(t_3 \cdot t'_3, -)$, $(t_4 \cdot t'_3, -)\}$. There are five informative tuples left: $t_1 \cdot t'_1$, $t_2 \cdot t'_1$, $t_3 \cdot t'_2$, $t_4 \cdot t'_1$, and $t_4 \cdot t'_2$. Let compute now the $entropy^2$ of $t_2 \cdot t'_1$ w.r.t. S using Algorithm 5. First, take $\alpha = +$ (line 1), then $S' = S \cup \{(t_2 \cdot t'_1, +)\}$ (line 2), note that there is no other informative tuple left (line 3), and therefore, $e_+ = (\infty, \infty)$ (lines 4-5). This intuitively means that given the sample S, if the user labels the tuple $t_2 \cdot t'_1$ as positive example, then there is no informative tuple left and we can stop the interactions. Next, take $\alpha = -$ (line 1), then $S' = S \cup \{(t_2 \cdot t'_1, -)\}$ (line 2), and the only tuples informative w.r.t.

S' are $t_4 \cdot t'_1$ and $t_4 \cdot t'_2$, we obtain $E = \{(3,3)\}$ (lines 6-10), and $e_- = (3,3)$ (lines 11-12). Finally, $entropy^2_S(t_2 \cdot t'_1) = (3,3)$ (lines 13-14), which means that if we ask the user to label the tuple $t_2 \cdot t'_1$ and any arbitrary tuple afterwards, then there are at least three other tuples that become uninformative. The computation of the entropies of the other informative tuples w.r.t. S is done in a similar manner. The *2-steps lookahead skyline strategy* ($\mathrm{L^2S}$) (Algorithm 6) returns a tuple corresponding to the "best" $entropy^2$ in a similar manner to $\mathrm{L^1S}$. In fact, Algorithm 6 has been obtained from Algorithm 4 by simply replacing $entropy$ by $entropy^2$. As we have already mentioned, the approach can be easily generalized to $entropy^k$ and $\mathrm{L^kS}$, respectively.

Algorithm 6 Two-steps lookahead skyline $\mathrm{L^2S}(D, S)$

1: **let** $E = \{entropy^2_S(t) \mid t \in D$ such that t is informative$\}$
2: **let** $m = \max(\{\min(e) \mid e \in E\})$
3: **let** e the entropy in $skyline(E)$ such that $\min(e) = m$
4: **return** informative t such that $entropy^2_S(t) = e$

6. EXPERIMENTS

In this section, we present an experimental study devoted to gauging the efficiency and effectiveness of the join inference strategies presented above. Precisely, we compare three classes of strategies: the random strategy (RND), the local strategies (BU and TD), and the lookahead strategies ($\mathrm{L^1S}$ and $\mathrm{L^2S}$). For each input database instance and for each goal query, we have considered three measures: the *number of user interactions* (i.e., the number of tuples that need to be presented to the user to label as examples in order to infer the join predicate), the *total time* needed to infer the goal join predicate, and the *time between examples*, using each of the above strategies as strategy Υ (cf. Section 5.1), and reiterating the user interactions until no informative tuple is left (halt condition Γ). Throughout the experimental section, by *join size* we intend the number of equalities from a (disjunctive) join predicate, while by *lattice size* we denote the number of nodes in the lattice of join predicates (cf. Section 5.2.1).

In our experiments, we have employed two datasets: the TPC-H benchmark datasets (Section 6.1) and a synthetic dataset we have built in the spirit of our introductory motivating example on Flight&Hotel (Section 6.2). The presented results cover the entirety of the TPC-H queries involving joins, as opposed to previous work [Bonifati et al. 2014a], which only focused on simple joins corresponding to key-foreign key relationships between pairs of tables. In fact, the queries reported in our work span an arbitrary number of tables. Moreover, to cope with the lack of disjunction in the TPC-H benchmark queries, we have built a synthetic Flight&Hotel dataset generator to be able to gauge our learning algorithms when disjunction is allowed. We discuss the results for the two datasets in Section 6.3 and we present the application of our techniques to interactive join query specification in Section 6.4.

It is also important to point out that in our experiments we focused only on learning settings where the input and output signatures coincide (i.e., on learning equijoins) and there are several reasons for this choice. First, we wanted to be able to fairly compare local and lookahead strategies (recall that the local strategies are meaningful only in such learning settings). Second, we recall that when the input and output signatures differ (i.e., for semijoins), there are cases where the problems of interest become intractable (cf. Theorem 3.5 and Theorem 4.6). Third, we observe that in addition to the intractability of the problems of interest there is another important issue that precludes in practice the learnability of semijoins, which is related to data visualization (that we also detail in Section 6.4). Since we focused only on settings where

the input and output signatures coincide, we were also able to apply a pre-processing technique in the spirit of Section 5.2.2 to remove redundant tuples before asking the user to label any tuple. Thus, from an initial large instance we have constructed a pre-processed smaller one that basically contains as many tuples as nodes in the lattice.

All our algorithms have been implemented in C. All our experiments were run on an Intel Core i7 with 4×2.9 GHz CPU and 8 GB RAM.

6.1. Setup of experiments on TPC-H

The TPC-H benchmark[1] contains the following eight tables (with the corresponding abbreviation in parenthesis): *part* (P), *supplier* (S), *partsupp* (PS), *lineitem* (L), *orders* (O), *customer* (C), *nation* (N), *region* (R).

Out of the 22 queries provided by the TPC-H benchmark, 20 exhibit join predicates. Since many of the queries contain aggregates, arithmetic expressions, groupby statements, etc. that fall beyond the scope of our learning techniques, and similarly to [Zhang et al. 2013], we modify the TPC-H queries by dropping all such operators while maintaining the join predicates. By performing this operation, some of the TPC-H queries actually become equivalent. Thus, we obtain a total of 15 different queries that we present in Table V. As an example, the first line of the table indicates that the TPC-H queries 4 and 12 have the same join condition "L[orderkey] = O[orderkey]" between the tables L and O. When a table appears more than once in a query, we use Arabic numbers to differentiate between these occurrences (e.g., N1 and N2 for query 7). Additionally, we saturate the join predicates by adding all join conditions that result by transitivity e.g., for query 21 we have "L1[orderkey] = L2[orderkey]" and "L1[orderkey] = L3[orderkey]" implies that "L2[orderkey] = L3[orderkey]". In Table V, we also illustrate the sizes of the considered joins, which span from 1 to 8.

In our TPC-H experiments, we used the instances found in the *ref_data* directory provided within the TPC-H benchmark to build the lattices on which the actual learning is done. There are eight such instances (that correspond to scaling factors from 1 to 100000), being of size in the order of MB. Basically, such instances are slightly different one from the other and, more importantly, they induce lattices of roughly same size, corresponding to at most 200 elements (we report these numbers in Figure 7). The main goal of our experiments was to check how many examples from the lattice should the user label in order to learn an arbitrary query. With such a goal in mind, it is important to pinpoint that the time for dataset pre-processing was not crucial for our analysis (the former being of the order of thousands (seconds) per query). Finally, since we observed high similarity across the results for the considered eight instances, and for the sake of conciseness, we only report the results for only the extreme and intermediate scaling factors i.e., 1, 100, and 100000.

We recall that our interactive strategies are ignoring the integrity constraints from the TPC-H schema and propose tuples to present to the user only by reasoning on the user annotations. The goal of such experiments on TPC-H is to evict the join predicates that rely on integrity constraints and that belong to the user's goal query. Nevertheless, attributes other than the ones involved in the keys and foreign keys may be involved in the join predicates as they exhibit compatible types. For instance, a value "15" of an attribute of a tuple may as well represent a key, a size, a price, or a quantity, etc. Actually, for a *ref_data* instance corresponding to a higher scaling factor, the number of equalities between such attributes exhibiting compatible types slightly decreases because the domains of the attributes are sparser. This explains the fact that the instances corresponding to larger scaling factors induce smaller lattices (cf. Figure 7).

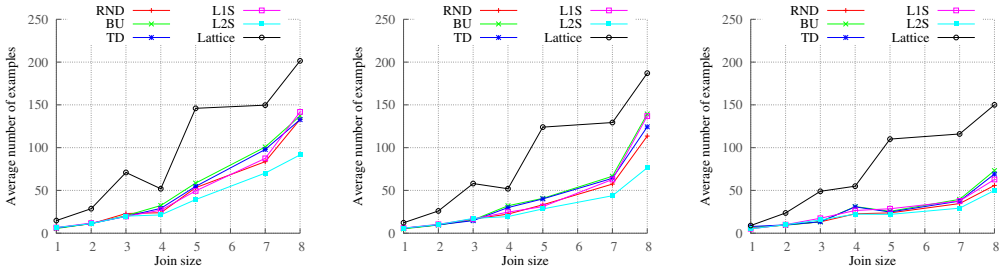
[1] `http://www.tpc.org/`

Table V. The TPC-H joins.

Query	*Tables*	*Join predicate*	*Join size*
4, 12	L, O	L[orderkey] = O[orderkey]	1
13, 22	C, O	C[custkey] = O[custkey]	1
14, 17, 19	L, P	L[partkey] = P[partkey]	1
15	L, S	L[suppkey] = S[suppkey]	1
11	PS, S, N	PS[suppkey] = S[suppkey] ∧ S[nationkey] = N[nationkey]	2
16	PS, S, P	PS[partkey] = P[partkey] ∧ PS[suppkey] = S[suppkey]	2
3, 18	C, O, L	C[custkey] = O[custkey] ∧ L[orderkey] = O[orderkey]	2
10	C, O, L, N	C[custkey] = O[custkey] ∧ L[orderkey] = O[orderkey] ∧ C[nationkey] = N[nationkey]	3
2	P, S, PS, N, R	PS[partkey] = P[partkey] ∧ PS[suppkey] = S[suppkey] ∧ S[nationkey] = N[nationkey] ∧ N[regionkey] = R[regionkey]	4
7	S, L, O, C, N1, N2	S[suppkey] = L[suppkey] ∧ L[orderkey] = O[orderkey] ∧ C[custkey] = O[custkey] ∧ S[nationkey] = N1[nationkey] ∧ C[nationkey] = N2[nationkey]	5
20	P, S, PS, N, L	PS[partkey] = P[partkey] ∧ PS[partkey] = L[partkey] ∧ P[partkey] = L[partkey] ∧ PS[suppkey] = S[suppkey] ∧ PS[suppkey] = L[suppkey] ∧ S[suppkey] = L[suppkey] ∧ S[nationkey] = N[nationkey]	7
5	C, O, L, S, N, R	C[custkey] = O[custkey] ∧ L[orderkey] = O[orderkey] ∧ S[suppkey] = L[suppkey] ∧ C[nationkey] = S[nationkey] ∧ S[nationkey] = N[nationkey] ∧ C[nationkey] = N[nationkey] ∧ N[regionkey] = R[regionkey]	7
8	P, S, L, O, C, N1, N2, R	L[partkey] = P[partkey] ∧ S[suppkey] = L[suppkey] ∧ L[orderkey] = O[orderkey] ∧ C[custkey] = O[custkey] ∧ C[nationkey] = N1[nationkey] ∧ N1[regionkey] = R[regionkey] ∧ S[nationkey] = N2[nationkey]	7
9	P, S, L, PS, O, N	S[suppkey] = L[suppkey] ∧ PS[suppkey] = L[suppkey] ∧ PS[suppkey] = S[suppkey] ∧ PS[partkey] = L[partkey] ∧ L[partkey] = P[partkey] ∧ PS[partkey] = P[partkey] ∧ L[orderkey] = O[orderkey] ∧ S[nationkey] = N[nationkey]	8
21	S, O, N, L1, L2, L3	L1[orderkey] = L2[orderkey] ∧ L1[orderkey] = L3[orderkey] ∧ L2[orderkey] = L3[orderkey] ∧ L1[orderkey] = O[orderkey] ∧ L2[orderkey] = O[orderkey] ∧ L3[orderkey] = O[orderkey] ∧ L1[suppkey] = S[suppkey] ∧ S[nationkey] = N[nationkey]	8

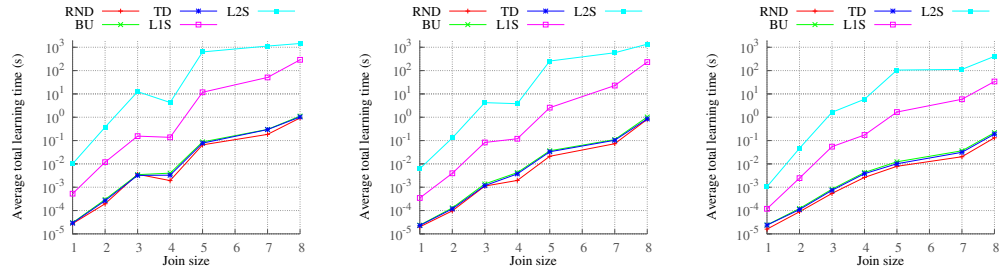
As already mentioned, the lattice of join predicates is dependent of the query that the user has in mind in the sense that the lattice is constructed based on the tables that the user wants to join. This is rather a natural assumption, since we assume that the user knows what tables are targeted by the query that she has in mind. The preprocessing transforms these tables into a lattice on which the actual learning is done. In situations where one relation appears more than once (as in some of the TPC-H queries), we simply characterize the different copies of such relation with appropriate aliases. Thus, we can accommodate join paths of arbitrary (albeit fixed) length, with multiple occurrences of the same relation.

Notice that there are two aspects of complexity of the lattice. One of them is the number of its elements, that we report in Figure 7 and changes from query to query since each query is formulated over a different combination of tables. The other aspect is the size of an element of the lattice i.e., the number of equalities that can be formulated over the given instance. For example, for our Flight&Hotel example, this number is rather small (i.e., 3), but in cases such as those where a relation is taken several times, the size of an element can easily become much larger (e.g., for the TPC-H query 21, where Lineitem is taken 3 times, there are over 200 equalities, out of which more than half are due to equalities between attributes of Lineitem coming from the



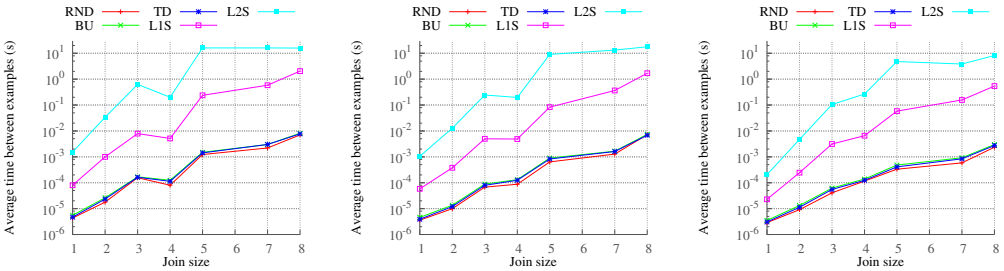
(a) TPC-H ref_data SF 1. (b) TPC-H ref_data SF 100. (c) TPC-H ref_data SF 100000.

Fig. 7. Average number of examples for learning TPC-H joins.



(a) TPC-H ref_data SF 1. (b) TPC-H ref_data SF 100. (c) TPC-H ref_data SF 100000.

Fig. 8. Average total learning time for the TPC-H joins.



(a) TPC-H ref_data SF 1. (b) TPC-H ref_data SF 100. (c) TPC-H ref_data SF 100000.

Fig. 9. Average time between examples for the TPC-H joins.

various occurrences of that table). In such cases, we do not construct the lattice such that it contains precisely one element per combination of equalities as described in Section 5.2.2 (that could go in the worst case up to 2^{200} elements in the aforementioned example), but we randomly extract a subset of it, that we fixed of size as large as the number of equalities (e.g., 200). In fact, we empirically observed that constructing such a subset is a reasonable choice since it allows the learnability of the goal query. Thus, a value of 200 as a lattice size actually means 200^2 possible Boolean values since each element of the lattice has also size of 200 equalities.

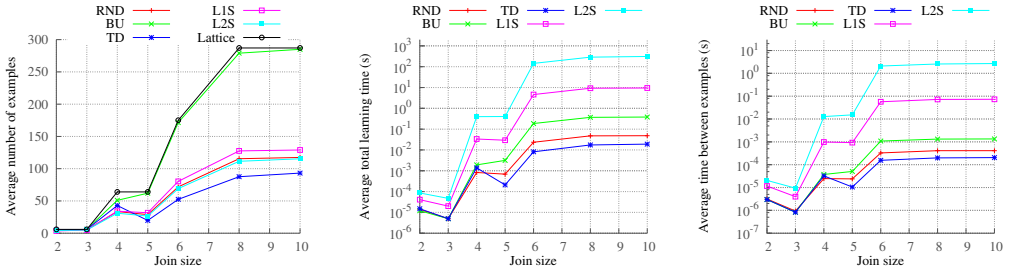
We present the experimental results on TPC-H in Figure 7 (the number of needed examples), Figure 8, (the total learning time), and Figure 9 (the time between two interactions). We discuss these experiments along with the synthetic ones in Section 6.3.

Table VI. The Flight&Hotel joins.

Query	*Tables*	*Disjunctive join predicate*	*Join size*
1	F, H	F.From = H.City ∨ F.To = H.City	2
2	F, H	(F.From = H.City ∨ F.To = H.City) ∧ F.Airline = H.Discount	3
3	F1, H, F2	(F1.To = H.City ∨ H.City = F2.From) ∧ (F1.Airline = H.Discount ∨ H.Discount = F2.Airline)	4
4	F1, H, F2	F1.To = H.City ∧ F1.To = F2.From ∧ H.City = F2.From ∧ (F1.Airline = H.Discount ∨ H.Discount = F2.Airline)	5
5	F1, H, F2	F1.From = F2.To ∧ F1.To = H.City ∧ F1.To = F2.From ∧ H.City = F2.From ∧ (F1.Airline = H.Discount ∨ H.Discount = F2.Airline)	6
6	F1, H1, F2, H2	F1.To = H1.City ∧ F1.To = F2.From ∧ H1.City = F2.From ∧ F2.To = H2.City ∧ (F1.Airline = H1.Discount ∨ F2.Airline = H2.Discount)	6
7	F1, H1, F2, H2	F1.To = H1.City ∧ F1.To = F2.From ∧ H1.City = F2.From ∧ F2.To = H2.City ∧ (F1.Airline = H1.Discount ∨ F1.Airline = H2.Discount ∨ F2.Airline = H1.Discount ∨ F2.Airline = H2.Discount)	8
8	F1, H1, F2, H2	F1.To = H1.City ∧ F1.To = F2.From ∧ H1.City = F2.From ∧ F2.To = H2.City ∧ (F1.Airline = H1.Discount ∨ F1.Airline = H2.Discount) ∧ (F2.Airline = H1.Discount ∨ F2.Airline = H2.Discount)	8
9	F1, H1, F2, H2	F1.From = F2.To ∧ F1.From = H2.City ∧ F1.To = H1.City ∧ F1.To = F2.From ∧ H1.City = F2.From ∧ F2.To = H2.City ∧ (F1.Airline = H1.Discount ∨ F1.Airline = H2.Discount ∨ F2.Airline = H1.Discount ∨ F2.Airline = H2.Discount)	10
10	F1, H1, F2, H2	F1.From = F2.To ∧ F1.From = H2.City ∧ F1.To = H1.City ∧ F1.To = F2.From ∧ H1.City = F2.From ∧ F2.To = H2.City ∧ (F1.Airline = H1.Discount ∨ F1.Airline = H2.Discount) ∧ (F2.Airline = H1.Discount ∨ F2.Airline = H2.Discount)	10

6.2. Setup of experiments on synthetic data

Since the TPC-H join predicates have only conjunctions, we have implemented a synthetic datasets generator to tweak our strategies also in the context of learning disjunctive join queries. We have generated tables of flights and hotels, in the spirit of our motivating example. More precisely, the relation Flight has 3 attributes (From, To, Airline) and the relation Hotel has 2 attributes (City, Discount). We abbreviate them by F and H, respectively. Moreover, when a table appears more than once in a query, we use Arabic numbers to differentiate between these occurrences (e.g., F1 and F2). In Table VI, we present the list of studied queries, which contain between 2 and 10 equalities. We have directly generated pre-processed instances based on the lattice. Their size depends on how many times each table is used in the goal query. In particular, in the considered queries (recall that we present them in Table VI), we may use (i) both



(a) Average number of examples. (b) Average total learning time. (c) Average time between examples.

Fig. 10. Summary of results for the Flight&Hotel joins.

F and H exactly once, (ii) F twice and H once, or (iii) both F and H exactly twice. These variations lead lattices from size 6 (that corresponds exactly to the one in Figure 3) to around 300. We plot these sizes in Figure 10. We illustrate the experimental results in Figure 10 for both number of examples and learning time, and we discuss them along with the TPC-H ones in Section 6.3.

6.3. Discussion

In this section, we discuss the experimental results for the two aforementioned settings.

First, we would like to point out that our implementation of queries and lattices is not specific to the problem of learning join queries. Thus, our strategies are applicable to any problem that can be reduced to finding informative nodes on a lattice, which can be represented with any general encoding. More precisely, for both datasets, we implemented a query as a Boolean array, having as length the total number of equalities that can be formulated over the given instance. For example, for our Flight&Hotel running example, there are 3 such equalities (From=City, To=City, and Airline=Discount, respectively), hence a query is a Boolean array of length 3. In particular, the query To=City $\wedge$ Airline=Discount can be seen as [0,1,1]. Then, a lattice is a collection of such Boolean arrays. For example, the lattice from Figure 3 can be seen as {[0,0,0], [0,0,1], [0,1,0], [1,0,0], [0,1,1], [1,0,1]}. Hence, as long as a problem can be reduced to finding informative nodes on a lattice, one can leverage our interactive strategies, which therefore have applications beyond the classes of queries studied in the paper.

The lattice size intuitively captures the complexity of an instance i.e., the larger it is, the more join predicates are in the lattice (cf. Section 5.2.1), and therefore, more interactions are needed to infer a join query on that instance. This explains the experimental results for the number of needed examples that we show in Figure 7 and Figure 10, respectively. We can observe that in the majority of cases TD and $\mathrm{L^2S}$ are better than the other strategies w.r.t. minimizing the number of interactions. However, none of them seems to be a winning strategy overall. In fact, their performance essentially depends on both the size of the join predicate (reported in Table V and Table VI) and the lattice size (reported in Figure 7 and Figure 10), but also on whether or not we allow disjunction in the goal query class.

It is important to point out that on all presented figures we included on the X axis the size of the goal join query, which spans between 1 and 8 for TPC-H, and between 2 and 10 for Flight&Hotel. In each graph, a value corresponding to a certain join size is actually obtained by averaging over all queries of that size. Thus, we can easily observe that a goal query of smaller size can be learned with less interactions. Such a trend is confirmed for each dataset.

Next, let us discuss how the number of interactions needed for each strategy depends on the lattice size. The essential insight is that $\mathrm{L}^2\mathrm{S}$ exhibits a better performance than TD with significant lattice size. A large lattice entails a higher number of join predicates in the lattice, and therefore, in such cases the inference requires more interactions. This explains the fact that for the most complex TPC-H joins $\mathrm{L}^2\mathrm{S}$ is the best strategy w.r.t. minimizing the number of examples. By opposite, with small lattices, and, consequently, fewer join predicates, the lookahead strategy might not be necessarily useful. Interestingly, when we also allow the disjunction in the synthetic experiments, we observe that $\mathrm{L}^2\mathrm{S}$ is no longer better than TD either. Intuitively, this happens because in the presence of disjunction the characterizations able to prune parts of the lattice are less aggressive than without disjunction (cf. Lemma 4.7 and Lemma 4.8, respectively). Thus, the entropies computed by the algorithm are smaller and choosing the best one among them is not necessarily better than a simple, local strategy.

Additionally, we observe that the total learning time for all strategies is always within a reasonable range, even though it may vary within this range with the different strategies. While for the random and local strategies the total learning time is always under a second, it can go up to at most a thousand of seconds for the lookahead strategies for the most complex TPC-H queries. However, such queries are the ones on which the lookahead strategy is most beneficial, which explains the difference in terms of learning time. As for the time between two examples, for TPC-H it is of roughly up to 10 seconds for two-steps lookahead, 1 second for one-step lookahead, and 0.01 seconds for local and random strategies. For the Flight&Hotel synthetic queries that also consider disjunction, the time between examples is always up to 3 seconds for two-step lookahead and less than 0.1 seconds for the others. Thus, this time is reasonably small for both datasets.

To summarize our experimental results, we point out that TD and $\mathrm{L}^2\mathrm{S}$ can be considered as a good trade-off between optimizing our antagonistic goals: minimizing both the number of user examples and the learning time. More precisely, if on one hand we have only conjunctions and a small lattice, or we allow disjunctions, TD is the best strategy; conversely, if on the other hand we have only conjunctions and more complex queries on dense lattices, $\mathrm{L}^2\mathrm{S}$ is the strategy to be chosen.

6.4. From query learning to query specification

As already mentioned at the beginning of the introduction, our study of query learning from examples is motivated by query specification for non-expert users. However, in this paper we are not focusing on interactions with real users, as we did in a system demonstration [Bonifati et al. 2014b]. The TPC-H datasets used in this paper are not appropriate for a user study, for which smaller and more bearable datasets would be needed in order to visualize the tuples. In our system demonstration, we mainly used two datasets: Flight&Hotel (as described in the running example of this paper) and sets of tagged images (e.g., the user can label pairs of images to learn queries like: select all pairs of images having the same color, shape or shading, etc.). Although the number of user examples for TPC-H can go up to 100, we point out that this upper bound corresponds to the most specific halt condition (i.e., presenting all tuples until no informative tuple is left). In practice, the user might want to stop the interactions earlier before this halt condition, and precisely, at the moment when she is satisfied with an intermediate inferred query. Such flexible halt conditions have been implemented in our system prototype and have been used with the customers of our demonstration. Finally, we would like to argue that what counts is the time to label the next tuple and that this time is reasonably small even for large datasets such as TPC-H (as shown in Figure 9).

Similarly to the experimental section of this paper, in our system we focused on settings where the input and output signatures coincide (i.e., on equijoins). As for the semijoins, we observed that in addition to the intractability of the problems of interest (cf. Theorem 3.5 and Theorem 4.6) there is another important issue that precludes in practice their learnability, which is related to *data visualization*. Notice that the data visualization does not pose any problem for equijoins since the user can visualize a tuple which contains all the information to permit her to decide whether she wants the tuple or not. However, we observe that this cannot be also applied for semijoins. A semijoin filters data from one source based on data from another source, hence showing a tuple over the output signature does not contain enough information to permit its labeling i.e., the user should also be provided with the second source, which can potentially be very large.

7. CONCLUSIONS AND FUTURE WORK

We have focused on the inference of join queries without the knowledge of referential integrity constraints. We have studied various settings, depending on whether or not we allow disjunction and/or projection in the queries. We have precisely characterized the frontier between tractability and intractability for the following problems of interest in these settings: consistency checking, learnability, and deciding the informativeness of a tuple. Then, we have proposed several efficient strategies of presenting tuples to the user and we have discussed their performance on the TPC-H benchmark and synthetic datasets.

As future work, we would like to investigate the problem of learning join queries in a context where the database instance is updated during the process i.e., the user may add new tuples to the instance and label them as examples. Another interesting direction for future work is to extend our approach to other algebraic operators. Additionally, our study makes sense in realistic crowdsourcing scenarios, thus we could think of crowd users to provide positive and negative examples for join inference. Finally, we think at employing our approach in data cleaning scenarios, where the user may label pairs of possibly conflicting tuples and then our algorithms could infer the conflicts that determined the user to label and a potential conflict resolution.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers, whose suggestions helped us to improve the presentation of the paper.

REFERENCES

S. Abiteboul, R. Hull, and V. Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

A. Abouzied, D. Angluin, C. H. Papadimitriou, J. M. Hellerstein, and A. Silberschatz. 2013. Learning and verifying quantified boolean queries by example. In *PODS*. 49–60.

A. Abouzied, J. M. Hellerstein, and A. Silberschatz. 2012. Playful Query Specification with DataPlay. *PVLDB* 5, 12 (2012), 1938–1941.

B. Alexe, B. ten Cate, P. G. Kolaitis, and W. C. Tan. 2011a. Designing and refining schema mappings via data examples. In *SIGMOD Conference*. 133–144.

B. Alexe, B. ten Cate, P. G. Kolaitis, and W. C. Tan. 2011b. EIRENE: Interactive Design and Refinement of Schema Mappings via Data Examples. *PVLDB* 4, 12 (2011), 1414–1417.

D. Angluin. 1988. Queries and concept learning. *Machine Learning* 2, 4 (1988), 319–342.

F. Bancilhon. 1978. On the Completeness of Query Languages for Relational Data Bases. In *MFCS*. 112–123.

G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. 2010. Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data. *TWEB* 4, 4 (2010).

A. Bonifati, R. Ciucanu, and A. Lemay. 2015. Learning Path Queries on Graph Databases. In *EDBT*. 109–120.

A. Bonifati, R. Ciucanu, and S. Staworko. 2014a. Interactive Inference of Join Queries. In *EDBT*. 451–462.

A. Bonifati, R. Ciucanu, and S. Staworko. 2014b. Interactive Join Query Inference with JIM. *PVLDB* 7, 13 (2014), 1541–1544.

S. Cohen and Y. Weiss. 2013. Certain and Possible XPath Answers. In *ICDT*. 237–248.

A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. 2010. Synthesizing view definitions from data. In *ICDT*. 89–103.

W. Fan, F. Geerts, J. Li, and M. Xiong. 2011. Discovering Conditional Functional Dependencies. *IEEE Trans. Knowl. Data Eng.* 23, 5 (2011), 683–698.

G. Fletcher, M. Gyssens, J. Paredaens, and D. Van Gucht. 2009. On the Expressive Power of the Relational Algebra on Finite Sets of Relation Pairs. *IEEE Trans. Knowl. Data Eng.* 21, 6 (2009), 939–942.

M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. 2011. CrowdDB: answering queries with crowdsourcing. In *SIGMOD Conference*. 61–72.

E. M. Gold. 1967. Language Identification in the Limit. *Information and Control* 10, 5 (1967), 447–474.

E. M. Gold. 1978. Complexity of Automaton Identification from Given Data. *Information and Control* 37, 3 (1978), 302–320.

G. Gottlob and P. Senellart. 2010. Schema mapping discovery from data instances. *J. ACM* 57, 2 (2010).

T. Imielinski and W. Lipski Jr. 1984. Incomplete Information in Relational Databases. *J. ACM* 31, 4 (1984), 761–791.

H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. 2007. Making database systems usable. In *SIGMOD Conference*. 13–24.

M. J. Kearns and U. V. Vazirani. 1994. *An introduction to computational learning theory*. MIT Press.

A. Lemay, S. Maneth, and J. Niehren. 2010. A learning algorithm for top-down XML transformations. In *PODS*. 285–296.

A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. 2011. Human-powered Sorts and Joins. *PVLDB* 5, 1 (2011), 13–24.

A. Nandi and H. V. Jagadish. 2011. Guided Interaction: Rethinking the Query-Result Paradigm. *PVLDB* 4, 12 (2011), 1466–1469.

J. Paredaens. 1978. On the Expressive Power of the Relational Algebra. *Inf. Process. Lett.* 7, 2 (1978), 107–111.

L. Qian, M. J. Cafarella, and H. V. Jagadish. 2012. Sample-driven schema mapping. In *SIGMOD Conference*. 73–84.

S. J. Russell and P. Norvig. 2010. *Artificial Intelligence - A Modern Approach*. Pearson Education.

T. Sellam and M. L. Kersten. 2013. Meet Charles, big data query advisor. In *CIDR*.

S. Staworko and P. Wieczorek. 2012. Learning Twig and Path Queries. In *ICDT*. 140–154.

B. ten Cate, V. Dalmau, and P. G. Kolaitis. 2013. Learning schema mappings. *ACM Trans. Database Syst.* 38, 4 (2013), 28.

Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. 2009. Query by output. In *SIGMOD Conference*. 535–548.

D. Van Gucht. 1987. On the Expressive Power of the Extended Relational Algebra for the Unnormalized Relational Model. In *PODS*. 302–312.

J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. 2013. Leveraging transitive relations for crowdsourced joins. In *SIGMOD Conference*. 229–240.

Z. Yan, N. Zheng, Z. G. Ives, P. P. Talukdar, and C. Yu. 2013. Actively Soliciting Feedback for Query Answers in Keyword Search-Based Data Integration. *PVLDB* 6, 3 (2013), 205–216.

M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. 2013. Reverse engineering complex join queries. In *SIGMOD Conference*. 809–820.

M. M. Zloof. 1975. Query by Example. In *AFIPS National Computer Conference*. 431–438.